Embedded C
–
Traps and Pitfalls

By

Chris Hills

3rd Edition
05 July 2005

Part 1 of the **QuEST** series

quest@phaedsys.org
http://quest.phaedsys.org/

Embedded C Traps and Pitfalls By

Eur Ing **Chris Hills** BSc, C Eng, MIEE, MIEEE, FRGS

Third Edition 3.2 05 July 2005

Updated as part of the **QuEST** series

Second Edition

presented at the
Embedded Systems Show and Conference, Olympia, London
24th May 2000 For Hitex (UK)
and (in condensed form)
ESS and Conference, Excel, London 16th May 2001 For Keil (UK)

First edition

presented at
JAva C & C++ conference, Oxford Union, Oxford UK Sept 1999
For the Association of C and C++ Users, see http://www.accu.org/

The slides and copies of this paper (and subsequent versions) and the power point slides will be available at http://quest.phaedsys.org/the authors personal web site.quest@phaedsys.org

This paper will be developed further.

Quality Embedded Software Techniques

QuEST is a series of papers based around the theme of ***Quality*** embedded systems. Not for any specific industry or type of work but for all embedded C. It is usually faster, more efficient and surprisingly a lot more fun when things work well. See http://quest.phaedsys.org

QuEST 0	Design and Documentation for Embedded Engineers
QuEST 1	Embedded C Traps and Pitfalls
QuEST 2	Embedded Debuggers
QuEST 3	Advanced Pain-free Embedded Software Testing For Fun

Q1	SCIL-Level
Q2	Tile Hill Style Guide
Q3	QuEST-C
Q4	MISRA-C compliance Matrix (several available)

Preface

The **QuEST** is a series of papers based around the theme of ***Quality*** **E**mbedded **S**oftware **T**echniques. It is not for a specific industry or specific type of work but for all embedded C. It is usually faster, more efficient and surprisingly a lot more fun when things work well.

This series will show you how to minimise the errors, the bugs and tiresome parts of software engineering (testing) and maximise the fun parts. With the right approach even testing becomes an interesting and fun challenge. Yes, I have done embedded system and unit testing!

The **QuEST** series is aimed at the "smaller" end of the market in most senses. The 8/16 market and the smaller companies and sub contractors. The larger companies usually have money for tools and procedures in place. The smaller companies in the current economic climate usually have fewer tools and smaller budgets. For these companies time really is money. So if you can complete the job faster, with fewer bugs, at a higher quality it really is more money in the pocket.

The **QuEST** series came about after I spent many[1] years in electronics, embedded software (8 to 32 bit), comms sw and hardware, specialist electronic production, avionics, installing ISO9000 a couple of times, joining the ISO-C and MISRA-C working groups and had a spell doing technical support. Being a member of ACCU also helped[2].

The technical support role was an eye-opener! Many times the same basic questions were asked. The same myths were repeated to me many times over. It was this that initially prompted me to write, in 1999, the first of the papers in this series and present it at a conference to try and dispel some of the myths and demonstrate software is *Engineering*. That was the Embedded C Traps and Pitfalls paper at an ACCU conference. The response to this paper along with re-writing and updating the Debuggers paper prompted this series as a homogeneous set.

The documents in this, expanding, series should lay the framework that will enable Engineers to do what they like best: design systems and write code.

It can be done! I have worked on projects that were completed on time, in budget with no over time. I have also heard of several others. I looked for common themes. I also looked at the research and statistics available. These are usually of little use to working Engineers but can show the trends of what works and what doesn't…. if you can get past the hype and the management speak

[1] "many" I worked it out and it is over 2 decades so I decided "many" as it was less depressing!
[2] I can trace it back even further to the influence of my father who designed aircraft engines in the 50's & 60's using computers when the computer's air conditioning plant had a building to itself and was the same size as the computer it cooled.

By the way, I am an Engineer, a working one at that! Not a consultant as I fail the basic requirement to be a consultant… I can't play golf!

The series looks at design, documentation, processes, coding standards and coding style, code construction and all the usual traps and pitfalls of [embedded] development. The "Embedded" is the important part as it does differ markedly from "normal" or desktop/mainframe development the series then follows on to the basics of debugging embedded software and systems before tackling the advanced testing and debugging methods.

The series is being constantly updated. I am looking at a major overhaul once a year. Adding new material as required and expanding topics that are there. Also of course correcting the errors and typos. In the initial years many typos were found and many helpful suggestions made. I am pleased to say virtually no technical errors were reported.

The author reserves the right to make typos and mistakes. However I would be grateful if anyone finding any errors, typos, mistakes etc would let me know. I will endeavour to correct them. Aly comments on anything presented, especially improvements would be good. Also any new ideas and additions would be welcome. Full credit will be given.

Chris Hills
05 July
2005
quest@phaedsys.org

Change log for 3rd edition

Jan 2003
From Gary A. Porter of Vicom Systems, Incorporated 47281 Bayside Parkway Fremont, CA 94538 phone (510)743-1164 email gary.porter@vicom.com A whole load of typos and helpful suggestions.(over 180 of them!! BTW yes, I am dyslexic :-) ***Effectively Gary did a very good job of proofreading. Many thanks.***

20-July-2002 Incorporated comments from Jonathan Kirwan (jkirwan@easystreet.com) on the use of libraries, the math's in particular and corrected typos that he spotted.

Contents

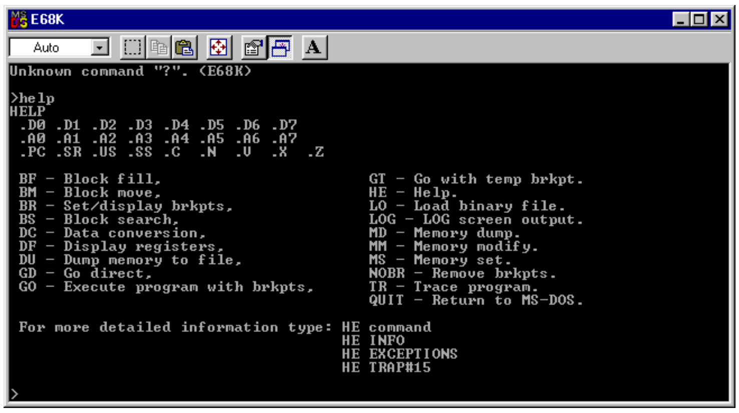
Embedded C - Traps and Pitfalls

1. Introduction

Back in the bad old days, which at the time of writing were less than 40 years ago, microprocessor programs were developed in assembler and blown into EPROM's such as 2708's, 2716's. With assembler, it was possible to know *exactly* what was in the EPROM. Know both in time (cycles) and byte by byte. Whether it functioned correctly was another matter! There was no possible way of knowing what the program was actually doing as it ran. Methods such as twiddling port pins, flashing LED's[3] or printf's to the serial port (if there was one) were used to tell where the program had got. Printf only worked on the larger micros as C was often not available on the smaller parts.

The lucky ones had ROM-based monitor programs. Originally these used an array of seven segment LED's and a hex keypad that allowed assembler to be single stepped and simple execution breakpoints to be set. At one time this was the height of sophistication! As time progressed a serial connection to a computer[4] or terminal allowed more flexibility. There were also some very rudimentary software simulators. A few, in the richest companies, were "blessed" with the ultimate tool: the In-Circuit Emulator (ICE). However, such was the initial cost and subsequent unreliability of some of the early ICE, that they were often ditched in favour of the more reliable ROM monitors. The one-thing emulators did have in common with monitors was a strong assembler-orientation.

SW development systems were, to say the least, were basic. Many things now taken for granted were not possible. For example: colour syntax highlighting, an IDE, simulator and debugger, project control and anything graphical.



```
Unknown command "?". <E68K>

>help
HELP
 .D0 .D1 .D2 .D3 .D4 .D5 .D6 .D7
 .A0 .A1 .A2 .A3 .A4 .A5 .A6 .A7
 .PC .SR .US .SS .C  .N  .V  .X  .Z

 BF - Block fill,                     GT - Go with temp brkpt.
 BM - Block move,                     HE - Help.
 BR - Set/display brkpts,             LO - Load binary file.
 BS - Block search,                   LOG - LOG screen output.
 DC - Data conversion,                MD - Memory dump.
 DF - Display registers,              MM - Memory modify.
 DU - Dump memory to file,            MS - Memory set.
 GD - Go direct,                      NOBR - Remove brkpts.
 GO - Execute program with brkpts,    TR - Trace program.
                                      QUIT - Return to MS-DOS.

 For more detailed information type: HE command
                                     HE INFO
                                     HE EXCEPTIONS
                                     HE TRAP#15

>
```

With great persistence, perspiration and a lot of ingenuity, usually using pencil on squared paper, working programs were produced. Whilst assembler is often specific to a particular micro, many programs required the same services, serial comms for example. For this reason people developed libraries. Re-use was around long before C++!

[3] LED's weren't around in the very early days so you used a 'scope on the pin.
[4] "computer" does **not** mean PC. This is some 10 years before the PC was born.

Incidentally the production of the reusable serial comms, terminal driver and boot loader modules were the initial steps in the concept of an operating system that was distinct from the application.

Embedded systems moved from assembler to C. There were other languages such as PLM, Forth, Modula-2 and Pascal along the way but C is by far the most common, most versatile and well known. Apparently, C is used in around 85% of embedded systems, with assembler used in around 75%. The assembler would be used in many C systems where accurate timing is needed and for some low level things like drivers. There are fewer assembler-only projects these days. Other languages are used but these rarely get over 10% market penetration.

If used properly, C is as robust and safe as any other high level language. That bears repeating:

When used correctly...

C is as safe and robust as any other high level language [Hatton].

As a spin off from the wide spread use of C in embedded systems there are many support tools, simulators, monitors and ICE that now support C source level debugging. This makes C an even more efficient way of producing embedded systems. I have been told that the C compiler is the most understood (and heavily tested) software on the planet. This in its favour.

Unfortunately, the History of C works against it. It is seen as a hacker's language and has a reputation as a read only language. Part of this is due to the obfuscated C competition to produce the most unreadable and tortuous, but fully legal, C program. I did have C program that was a single (long) line. The main() function had an empty pair of braces{}. It would (without a word of the text visible in the program) compile and print out to screen the whole of the 12 days of Christmas from the very strange executable parameters!

The other problem is that, just as everyone thinks they can write a book many think they can write a C compiler or debugger. Unfortunately, this means that there are also a lot of poor quality tools (and a lot more poor quality books) out there. Choose wisely.

As an example, I was once asked to set up (for a UK University) a cross compiler. It was a PC-hosted Modula-2 68K compiler. However, upon trying to install it I ran into problems. The compiler, it turned out, had been written in assembler. The company who produced it had virtually no design documentation, only the users manual. There was no test suite or proof of testing. The libraries were for an Atari ST running an OS (TOS): there were no stand alone embedded libraries (even though this was sold as an embedded cross compiler) As the programmer had left the company (and the country), they were not able to help at all.

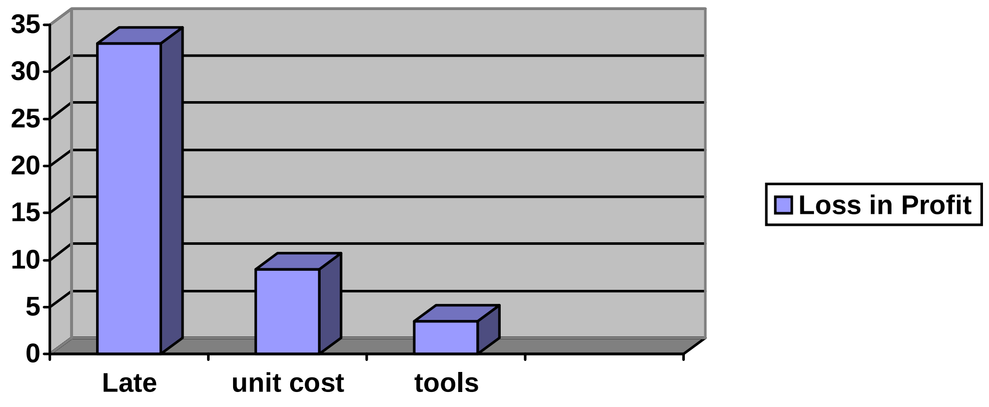
Thus, a cross compiler for safety critical use with a "safe" language (Modula-2) was in fact a totally unsafe piece of software. Had I got the compiler up and running the University would have had no idea how unsafe the underlying construction was. It would have been used on several safety critical projects because Modula-2 was a "safe" language.

There are many more C tools out there than Modula 2 ones. Therefore, there are likely to be many more quality C tools out there. However you should still be very careful, as there will also be many more poor quality C tools out there. The advantage with the Internet is you can find many useful tools quickly and get hold of the authors as well. Make sure of the pedigree of any embedded development tools you buy. As any mechanical, civil, aeronautical or electrical Engineer will tell you:

It is well worth buying good quality tools.

I have a very interesting graph that proves the point very well. The graph shows that (all other things being equal) that for a product with a 5 year life the following was the average over a large number of projects surveyed. If you are 6 months late to market you can loose a third of your net profits!!! If you are 9% over budget in production costs you can see a loss of 20% in your profits. However a 50% increase the cost of the development tools could hit your profits by as much as 3%!

Loss in Profits after Tax.

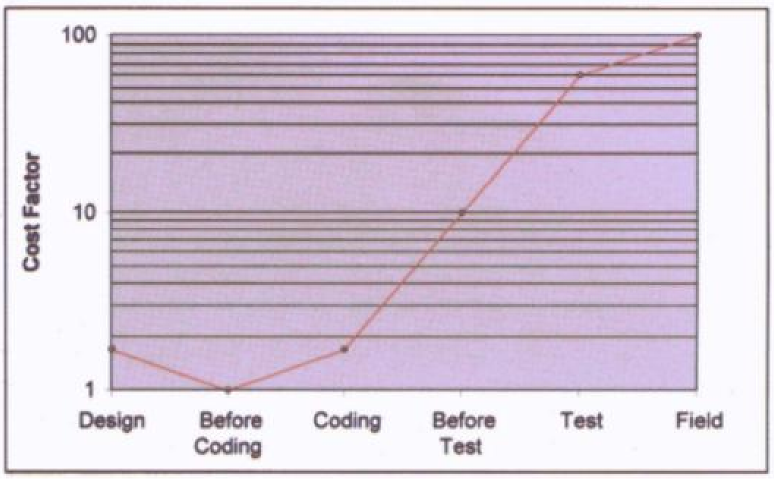


This assumes 20% growth rate, 12% annual price erosion and a 5-year product life cycle.
Source: Kcinsey & Co.

As you can see getting the right tools in may be a large hit in the development budget but it will pay dividends in the long run. In addition, the tools will be usable on the next project thus giving even greater savings or at least minimising the potential losses.

For this paper, I am only interested in the production of good C source code. I am not looking at how you got your design, CASE tools, how the teams were organised or anything of that nature. This is largely because the majority of embedded systems (8 and 16 bit) are on the small side and do not warrant computerised CASE tools. As Les Hatton once said "Without proper use of the tool a CASE tool can create a mess far more effectively and efficiently!" The *correct* use of a CASE tool can speed up a design and reduce errors. However, the majority these days are aimed at OOP, C++ and Ada. There are few CASE tools aimed at the smaller C projects.

As this graph here shows the cost of fixing a bug rises as the life cycle progresses. The idea behind this paper is to highlight ways in which you can lower this graph in the coding and "before testing" phases.



The second paper in the **QuEST** series: **Embedded Debuggers** (see quest.phaedsys.org) deals with the debugging of the code once you have got it (almost) running on the target. This will help you on the transition to testing and choosing the appropriate testing tools.

The third paper in the **QuEST** series: **Advanced Embedded System Testing For Fun** (also on quest.phaedsys.org) goes on to cover testing, automated regression testing and what to spend your bonus on in your increased spare time, having got the project in before time! This should help lower the curve in the test and field phases. The cumulative effect will be to produce fully tested and debugged code far faster and a lot more efficiently. This gives the SW engineer more time to enjoy the project.

In many cases discussed it is the type of tool not the make of tool that is important. For example static analysers run from Free to £6,000 each. However, the most cost effective one for my money costs £127. You will need to use one but the nature of your project will determine which type of static analyser is the most appropriate. Note for static analysis you *MUST* use one. The question is which!

Why "must"? Well, there are some good technical reasons I shall come on to later. There are some good commercial reasons, which I shall also come on to later. Then there are the legal reasons, which the government will come on to later….

At this point I have to state that whilst I am Eur Ing C. Hills BSc, C. Eng, MIEE, FRGS etc. none of that lot relates to law and I am NOT in any way legally qualified. ***The following is purely my opinion.***

The is an amendment proposed to the Manslaughter Act called "Reforming the Law On Involuntary Manslaughter: the Government's Proposals" I found it at http://www.homeoffice.gov.uk/consult/invmans.htm

This is the infamous "corporate manslaughter" amendment. It talks of the Herald of Free Enterprise disaster, the Kings Cross Fire and the Clapham Rail crash… The point is that it will be possible to sue the entity of ***a company*** (not a person) for Corporate Manslaughter. In the case of the Herald of Free Enterprise it was suggested that whilst no one person was guilty of a specific act which caused the disaster, it was due to the way ***the company*** ran the ship. Thus the company could be sued for Corporate Manslaughter.

There are, in my opinion, unfortunately a couple of false hopes in the amendments. First it is not backdated. It will only affect accidents after the date it comes in. This was (when I asked the Home office in late 2001) expected to be in the 2004 Session of parliament.

Why is this a false hope? Follow this logic. The new rules come in during 2005. There is an accident with say a car in 2007. The makers are sued under the corporate manslaughter rules…. The car makers say it was WXYZ's sub-system…. WXYZ say it was the SW written by A-Subbie Ltd. The problem was this was the sw you wrote in 2002…. This (in my opinion) could be looked at under Corporate Manslaughter. "Not retrospective" means that you cannot claim "Corporate Manslaughter" on ***accidents*** before the amendments come in.

Most of the systems involved in accidents after the amendments are in will have been developed before this date.

Can you show due diligence? Can you show that you engineered the SW using good methods and approved principles, the correct tools that were up to the job?

The other false hope is the words "corporate manslaughter" only appears in a few places. Most of the acts to be amended such as the Railways Act, the Aviation and Maritime Security Act, the Bail act, The Criminal Law Act etc only mention the addition of "Reckless killing" and/or "killing by gross negligence". . The one place where the phrase "Corporate Killing" does appear is in the Coroners Act.   As far as I can see only a coroner can invoke "Corporate killing"…

This in my opinion may give a false hope. Remember the coroner "only" investigates when there is a dead body to establish cause of death. A Coroner would get involved for example in a rail crash, an aviation or maritime accident. This, in my opinion, would be enough for a civil action if not a criminal one whenever some one is killed.

The government paper defines corporate manslaughter as

```
(1)   A corporation is guilty of corporate killing if-
```

```
     (a)  a management failure by the corporation is the
        cause or one of the causes of a person's death:
        and
     (b)   that failure constitutes conduct failing far
        below what can reasonably be expected of persons
        of the corporation in the circumstances.

(2)  For the purposes of subsection (1) above-
     (a)  there is a management failure by a corperation
        if the way in which its activities are managed or
        organised fails to ensure the health and safety of
        persons employed in or affected by those
        activities; and
     (b)  such a failure may be reguarded as a cuase of a
        person's death notwithstanding that the immediate
        cause is the act or omission of an individual.
```

This is from the proposals available from HMSO and *http://www.homeoffice.gov.uk/consult/invmans.htm* From what I can see the intended punishment is a fine rather than sending the directors to prison. However As I have made clear I an NOT in anyway legally qualified and readers should look at the proposals themselves and seek qualified legal opinion.

Another interesting comment I cam across sin an email will concern many:-

> I come from an industry (medical devices) where good documentation is required and highly structured, so it's not "ALWAYS bad", but I do know what you mean, and most of the emotional side is usually picked up by the lab notebook which is what you refer to as a "diary". Because not all design decisions are done for practical reasons. ***The only problem with using a lab notebook in this fashion is how the notebook could be used later in a court of law.*** Product liability is a serious consideration for us in the U.S.

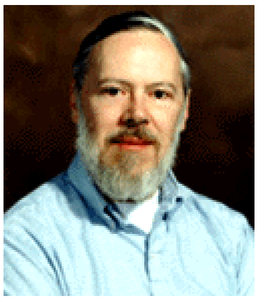
This says that if you don't have full and complete documentation and there is a court case your lab not book could end up in court as the documentation… Look back in your current notebook and see if you would be happy with that in court! Especially as it is up to the judge to decide what is relevant and he might see things entirely differently to you. The other thing you might like to worry about is that emails are increasingly used in court. I know they will probably be ruled as inadmissible but that is after they have been seen by both sets you lawyers.

The other reason for doing things correctly is that over time they have been shown to save time and money.

The ART in Embedded Engineering comes through engineering discipline

2. History

The problem with C is its history. I do not propose to re-tell "The K&R Story" [K&R] here. However, there are some parts pertinent to this paper.



I recommend that people read the paper by Dennis Ritchie [Ritchie] this is available from his web site: http://cm.bell-labs.com/cm/cs/who/dmr/index.html

C was developed initially (between 1969 and 1973) to fit into a space of 8K. Also C was designed in order to write an (portable) operating system. Unlike today, where disks and memory are inexpensive, at the time Multics was around operating systems had to take up as little space as possible, to leave room for applications on minimal memory systems. This makes it ideal for embedded systems.

C was developed from B and influenced by a group of several other languages. Interestingly BCPL, from which B was developed used // for comments just as C++ does and now finally C99!

One of the problems with C is that now the majority of people learn C in a Unix or PC environment with plenty of memory (real or virtual), disk space, native debugging tools and the luxury of a screen, keyboard and usually a multi-tasking environment.

Because C was originally designed for (compact) operating systems it can directly manipulate the hardware and memory addresses (not always in the way expected by the programmer). This can be very dangerous in normal systems let alone embedded ones!

C permits the user to do many "unorthodox" things. A prime example is to declare 2 arrays of 10 items A[10] and B[10]. Then "knowing" that (in the particular implementation in use) they are placed together in memory use the A reference "for speed" step from A[0] to A[19]. This is the sort of short cut that has got C a bad name. Yes, I have seen this done.

The syntax of C and its link with UNIX (famous for its terse commands) means that many programmers try to write C using the shortest and most compact methods possible. This has led to lines like:

```
while (l--) *d++ = *s++;
```

or

```
typedef boll (* func)(M *m);
```

This has given C the reputation for being a ***write only*** language and the domain of hackers.

As C was developed when computing was in its infancy and there were no guidelines for SW engineering. In the early days many techniques were tried that should by now have been buried. Unfortunately, some of them live on.

2.1. From K&R to ISO-C99 :- A Standard History of C

In the beginning in 197... Well it starts in the mists of legend... The best social/technical description I have seen is the paper **The Development of the C Language by** *Dennis M. Ritchie* it is (as of early 2001) available as a pdf http://cm.bell-labs.com/cm/cs/who/dmr/index.html. (If you have any problems finding it contact chris@phaedsys.org) This dates the beginnings of C as "about" 1969 to 1973 depending how you measure it. C evolved from B and BPCL when (modern) computing was only about 20 years old and microprocessors had yet to be invented. This paper is well worth reading as, in my view, it gives the best description of how it all came about (and why). Do not expect to learn C from this paper.

BTW Unix was so called because it was a ***Single User*** OS… a parody of Multics the multi user OS that they had. No, it was not run on a PDP11 first but a PDP7. Not a lot of people know that!

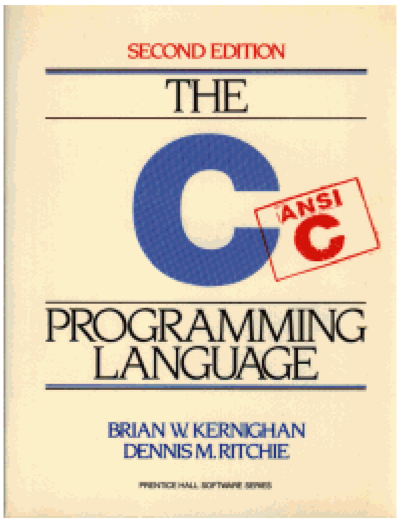
2.1.1. K&R (1st Edition) 1978

1978 saw the publication of The C Programming Language by Kernighan and Ritchie, thereafter known as "K&R". This was The Bible for all C programmers for over a decade. Unfortunately, many still cling to the faith despite the language changing a lot in the intervening 25 years. Even Dennis Richie said of K&R 1 "*Although it did not describe some additions that soon became common, this book served as the language reference until a formal standard was adopted more than ten years later."* See his paper cited above. This comment from one of the authors dents the mantra of many disciples that The Book is The Definitive Reference! K&R later published a new edition "K&R 2nd Ed" in line with ANSI C 1989.

Something else should be borne in mind when reading K&R 1st edition. It was written by experienced operating systems programmers for experienced UNIX programmers (by this time UNIX was a multi-user, multi-task OS). K&R is not, and never was; an introductory text on C for novices let alone 8 bit embedded systems programmers.

Having debunked K&R 1st Edition one should heed the commandment (found in some form in most faiths) "honour they parents." K&R 1st edition is the root of C and the source from which it all flowed. If you can find a copy (or K&R 2nd edition) buy one and dip into it but do not use it as a definitive reference or use it to teach people. Many (ten or twenty years ago) did learn from it, but then, it was the definitive (and only) work.

2.1.2. K&R (2nd edition 1988)



K&R 2nd edition gives the syntax changes and "improvements" in C over the decade since K&R1 and it brought K&R into line with the ANSI C 1989 standard. If you want a K&R for practical use this is the edition to have. You should remember it is not the definitive as from 1999. I expect there will not, despite public pressure, be a K&R3 as all the authors have moved on to new things in the last decade. (The authors previously have stated that there would *not* be a K&R3 but in early 2001 they left the door open…) Note the standard takes longer to ratify and publish that a book, which is why K&R (who were part of the US (ANSI) ISO committee anyway) got their book out ahead of the standard.

2.1.3. ANSI C (1989)

Eventually in 1989, due to the large number of people using C ANSI produced a USA standard that became the de-facto world wide standard until 1990. This stabilized the language and gave everyone (except Microsoft) a standard with which to conform.

2.1.4. ISO-C90 (1990)

ISO/IEC 9899 Programming Languages-C
At the end of 1989 ISO (and IEC) with all it's committees from many countries world wide adopted and ratified the US ANSI standard as an International Standard. From this point in theory, if not in practice ISO-C superceded ANSI C as the definitive standard. However, it should be noted that the only difference between ISO and ANSI C during the 1990's was the Chapter numbering. One of the standards had an additional chapter before the actual standard throwing all the chapters out by one. Paragraph numbering was the same in both.

NOTE:- This version of ISO C is used for MISRA-C also for most embedded compilers as later "improvements" such as multi-byte characters and other changes for C99 were not needed (and in many cases not easy to implement) . At the time of writing , early 2003, there were still only two C99 compilers available.

ISO-C Amendment 1 1993
Multi byte Characters

ISO-C Technical Corrigendum 1995/6 (T1)

Work on the new standard starts.

Due to the fact that many things (eg MISRA-C) reference ISO C 90 the author has managed to persuade BSI (British Standards Institute) to make ISO C90 (with Amendment 1 and TC1) available again at a comparatively low price of £30 (about $45US).

2.1.5. ISO-C99 ISO/IEC 9899:1999

The ISO-C99 is now the definitive international work on the Language… It is not what I would call "readable" though. It was some months after the ISO-C99 was finished that ANSI (and all the other National Bodies around the world) adopted it.

A copy of ISO-C is a useful document to have (if only to win bets at lunchtime!) ISO-C99 ISO9899:1999 This can be obtained (correct as of early 2001) for $18 US as a PDF from :- www.techstreet.com/ncitsgate.html which is where I (and most of the UK standards panel) got my copy. It prints out to 537 pages. I printed it out, on a double-sided photocopier via the network on A4 double sided and it is quite usable.

The good news is, at the time of writing (November 2001) It is likely that a book publisher in partnership with the ACCU (see www.accu.org) will turn both the C and C++ standards in to books at around the £30 mark!

2.1.6. ISO/IEC 9899:1999 TC1 2001

The link leads to a seven page PDF document of 118062 bytes containing ISO/IEC 9899:1999 TECHNICAL CORRIGENDUM 1 Published 2001-09-01

http://ftp2.ansi.org/download/free_download.asp?document=ISO%2FIEC+9899%2FCor1%3A2001

The TC is freely available and the link abouve should download the PDF directly.

2.2. The Future: Back to C. (Why C is not C++)

The main problem at the moment is that as of November 2001 (nothing had changed by Jan 2003) no one has implemented a full C99 compiler for embedded use. There are dark mutterings in the embedded world that they may stay with C90.

Many people ask for C++ on small-embedded systems. What most people do not realize that whilst C++ was *developed from* C the two are now separate languages. In the early days C++ was a superset of C. This ceased to be true from the mid 1990's, both languages have moved on with slightly diverging paths.

C++ is being used on the desktop, 64, 32 and some 16 bit systems under UNIX, MS Windows and a variety of high end embedded RTOS. 2000 saw the start of some embedded C++ for 16 bit systems but that is as far as it will go. The use of C on the desktop has declined and the majority use is now in embedded systems often without an RTOS. After I wrote this some while ago I have been corrected that many compiler writers and systems writers also use C.

In late 2000 the ISO C committee was getting more work packages to do with embedded C and things to help the conversion of mathematical Fortran users to C. It was at this point the editor of this work took over as Convener of the UK ISO C committee.

The next round of work was meant to help the embedded user and will move C further from C++. Unfortunately the amendments were mainly in the form of DSP math and extensions only of use to 32-bit embedded systems with lots of space. There was a lot of discussion in 2003 with some violent disagreements of the direction C should take.

New features in C++ that might once have been put into C are less likely to happen. Partly because there are more embedded people involved and there are fewer desktop people involved. The other reason is that some C++ is not possible in 8 bit systems. There are some C++ compilers for small systems but the are not widely used and have some severe restrictions. Their use is dictated more by fashion than engineering reasons. In fact even C++ is being restricted as EC++ for embedded use. For embedded C++ see:- http://www.caravan.net/ec2plus/ Where you can get the Embedded C++ "standard" as supported by many compiler manufacturers. This was an initiative started in Japan that has speard world wide.

The other major thing the (UK) standards panel is trying to do is stabilize and iron out the ambiguities of the C99 standard. The ambiguities are one of the reasons that, two (now three) years later, no-one had managed to do a fully implemented C99 Embedded C compiler. Actually I don't think there is a full C99 compiler for any use. As of the summer of 2002 a couple of compilers had managed it but no mainstream industrial embedded compilers vendors even thought about it.

Where next for the C standard? Judging from history, you should expect preparation of the next revision of the C standard to begin around 2004. Most likely we will ask for feedback on an early draft around 2007. In between those times is the best time to provide constructive input, but be warned that unsolicited proposals without an active champion participating in the committee are unlikely to get very far. If you really want to work on substantial improvements, it would be wise to join the committee (via your National Body) well in advance, so you can gain a feel for how the group dynamics work. If you want to get involved please email me at chris@phaedsys.org

2.3. What to read for Embedded C?

There are thousands of C books out there… Few are *really* good. Most are for the desktop (MS & MAC) and Unix. For a good source of book recommendations try the ACCU at www.accu.org. They have independent reviews of over 2000 C, C++ and SW engineering books. They do not sell books so the reviews are completely independent and written by working Engineers. There is one infamous review that starts "I did not pay money for this book and I would suggest that no one else should either…"

There is a list of books in the reference appendix. However, remember: Most C books are written for the desktop programmer not for embedded systems. I would still get K&R 1st edition as a *historical reference* but not as a first C book to learn from. I have an ISO C standard but that is not a book to learn from either! You do not buy a dictionary to learn how to write novels.

One thing to be wary of is that if the book is written by an academic it is likely to have been written for his course… It may well refer to development boards and other equipment made by him at the university and not generally available. Also it may assume you are doing or have done other courses and modules in the collage and therefore miss out useful information because you will get it on the other course. Not all books written by academics are like this but do take care when buying.

Incidentally if anyone wants the ISO C99 standard the best place to get it from is a US web site www.techstreet.com/ncitsgate.html

Note:- At the time of writing (summer 2002) BSI were looking at publishing the ISO C and C++ standards at £30 printed but loose leaf. At the moment this looks like a good deal unless you have access to [someone else's] printer that can print double sided and will run off 550 pages for free.

3. SW Engineering with C

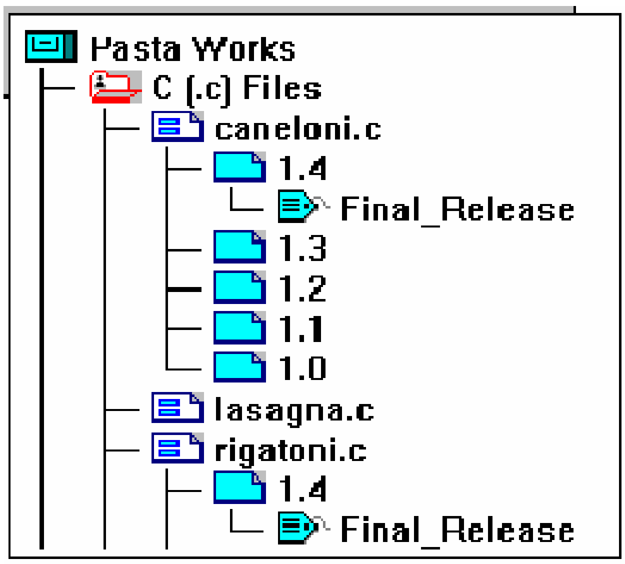
As I and many others [Hatton][Misra] [COX] [Pressmann] have said, *when used properly*, C can be as safe as any other High Level Language. For embedded use there are some additional things one must think about. This paper, in looking at embedded C, will also cover many things that will be of use in general C programming.

As mentioned in the introduction I am only looking at the production of safe, robust C source code. How you got your design, - pencil and envelope (50p) or CASE tool (£5,000) - is not relevant here. Neither is the design method, though there should have been one. They will be covered in Quest 0… In true Star Wars fashion of part 1 arriving decades after part 3: Quest 0 is being researched now for publication in late 2003 some 4 years after parts 1-3!

3.1. Organise

First, organise your files. Both the code files and documentation. Version Control (or Revision Control System, RCS) has been around for many years, yet large numbers of engineers still do not use it. There are basic RCS/SCCS systems that run on a one-machine, one-user basis up to the systems that can track files across linked networks and the Internet for projects strung across many counties in several continents. I have worked on one of these where there were two engineers whose prime task was to administer the VCS . Appropriately the system was a large communications controller.

What is VCS? It is basically a database that will hold all the versions of a file. Thus when bugs are fixed or other changes made to a file both the original and the new versions can be stored and retrieved.



```
Pasta Works
└─ C [.c] Files
   ├─ caneloni.c
   │  ├─ 1.4
   │  │  └─ Final_Release
   │  ├─ 1.3
   │  ├─ 1.2
   │  ├─ 1.1
   │  └─ 1.0
   ├─ lasagna.c
   └─ rigatoni.c
      └─ 1.4
         └─ Final_Release
```

As can be seen in the diagram there are five versions of caneloni.c

Most VCS systems permit the labelling of file version. In this case V1.4 is "Final Release"

The system will have the ability to retrieve all files associated with a label.

Usually multiple labels can be assigned to a file. Thus a standard module can be used in several projects. So a standard comms module could be labelled "Release_1", Release_2" and "Special_2"

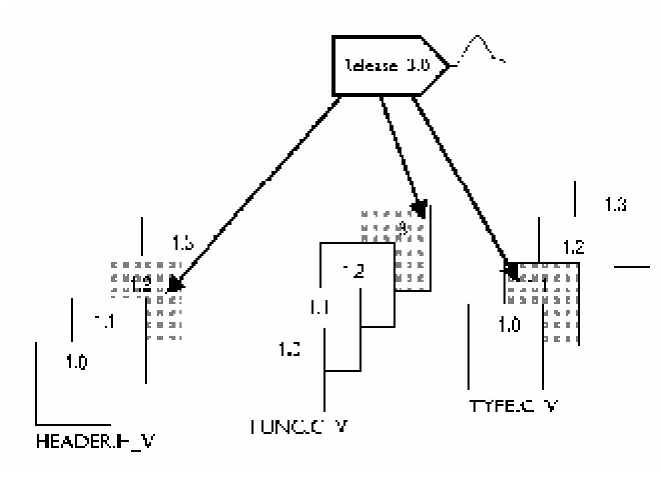
When the VCS is linked with a make system it gives the ability to "make Release 1".

Most compiler IDE's will now seamlessly integrate into a VCS.

VCS means that you never lose a file and can recover any version of a file and therefore create any version of the software (as long as the files have been checked in). This can be very useful when major changes are added to a file for the wrong reason and need to be removed.

It also means that when there is a panic because you suddenly need a copy of V1.34 (current version V5.601) you can blow an EPROM to send it out to a customer because you promised four years ago to support the version he had for the next five years!

The other very good use of VCS is that it permits developers to get on with the next version without affecting the current builds. IE one can set up the VCS to let the test team get the "Release" version of the file released for testing. Then there is the choice of fixing the bug in the version the developer is working on or branching the file to give another copy (still tracked by the VCS and linked to the original file). Most VCS systems permit the merging of branches later. This is usually a semi-automatic procedure. Therefore you can bug fix the current released version and feed the problems in to the new version to be fixed in the most suitable way. IE some new functionality may have removed the bugged code anyway.



VCS also stops two people accidentally working on copies of the same file. It usually requires several intentional acts (bypassing passwords and warning dialogue boxes) to get a second write-able copy out and several more intentional acts to check it back in in-place of the original version pulled out. Often it will require the intervention of the VCS administrator. SO it will stop accidents but still permit the bypassing of the system in emergencies. It is all, of course, fully logged and the files can all be recovered.

Many modern C development systems have hooks in them to interface to the VCS systems so that once set up they become transparent to the developers.

Due to automatic time and date stamping in VCS systems, no matter how slack you, or some one else gets, you should be able to (this is the part ISO 9000 people like) show a complete audit trail from the day the file was created. So not only will you be ISO9000 compliant, it will save you many hours when someone suddenly needs an old version. As a benchmark in late 2001 VCS costs ranged from FREE to the average price of £400 per seat with a few top end systems costing a little more.

VCS systems cover all sizes of project. I have seen an extreme case where a very large embedded project consisted of several linked systems produced by a couple of hundred engineers across 9 sites in 6 countries on three continents. The VCS system (Clear Case from Pure Atria) could not only track all the files but also synchronise all 9 of the databases automatically. This meant that all the developers and testers were always working on the correct (but not necessarily the same) versions of software. This system was able to cope with two changes of target CPU architecture! All the reusable Sw modules were kept and moved (with their history) to the "new" project.

At the other end I have used a simpler system (PVCS from Merant) that I found to be very useful on a single machine (or small networks) running one or several projects. The PVCS suite can also integrate make and bug reporting modules to give a full ISO9000 and CMM compliant system complete with audited and documented bug fixes, builds, manifests etc.

These systems do cost money and take time to set up, but are worth their weight in gold when a customer wants a mod done to a project you last worked on 2 years ago. The other nightmare scenario is where the customer wants a mod and the code has since been modified for something else that the customer not only doesn't want, but refuses to accept.

So, we now have a project where we have organised files where we can get at any version and easily build any version of the system. Incidentally, this also helps with testing as test scripts can be held in the VCS in the same way and any test suite rebuilt. The only thing to watch out for is these systems store deltas of text files. Most let you baseline and start a new set of deltas but for the storage of non-ASCII files they usually make complete copies. This takes up a hell of a lot of disk space so be warned!

What we now need is something in the files we have organised. Before leaving the RCS completely there is one last point that goes into the next section. The RCS systems can usually insert into the source files things like current version, change log, file names, paths to archive, author etc. In the example shown (PVCS) it is the text between the "$" delimiters. This insertion is automatically done by expanding these keywords. In this example the whole history block after the $log is also added automatically.

This automatic insertion means that a simple template is all that is needed for modules. The developers do not need to complete it (well only the odd line) as the VCS system does it for them.

In the example below, the file name, author, revision, and history log are automatically inserted. So even with the tardiest of developers an ISO9000 audit trail is automatic. As the log uses the login name specific to the user it will be obvious who did not correctly complete the header block.

```
/*****************************************************
** $Workfile:   U1CO0001.C  $
** Name: Application Block
** Copyright :PhaedruS SystemS 1999
** $Author: Chris Hills$
** $Revision:   1.1  $
**
** Analysis reference:123/ab/45678/001 5.6
** Input Parameters:  NONE
** Output Parameters: NONE
**
** $Log:   C:/ENG/KOS2/A2C001.C_V  $
**
**   Rev 1.1   06 May 1998 16:48:28   HILLS_CA
**Issued for review
**
**   Rev 1.0   01 Apr 1998 13:09:02   HILLS_CA
**initial version
**
*/



/********** End of $Workfile:   U1CO0001.C  $ *************/
```

A full style guide, ***The Tile Hill embedded C Style Guide*** will be available at quest.phaedsys.org

4. Good C

Now having organised all the files we need "Good C" in them. What is good C?

> It must not contain errors
> It must perform as expected.
> It must be repeatable
> It must be easily and clearly readable

These seem simple enough and may initially appear to overlap.

Firstly, the C should not contain any syntactical or semantic errors. This is not always as obvious as one might think. Syntactical errors the compiler picks up but semantic ones can be far subtler. They can also be a lot more difficult and time consuming to find if left to the test and debug phase to find. The cost of fixing an error rises almost exponentially the longer it is left before finding it. A bug found at the time it is coded costs very little to fix but thousands of pounds and days if it gets into the field.

The syntactical and semantic errors should be found as the source code is written using static analysis, not the compiler. The compiler is a translator not a test tool. Having said that the compiler should always be set to the highest level of warnings.

The most cost effective way of removing these errors and warnings is static analysis. This looks at the code without compiling or running it. It can find a large number of errors and possible errors (Warnings) at the time the code is written. PC-Lint for example will integrate into most compiler IDE's and can be used to check the code as it is written.

After the syntactic and semantic errors are removed does the code do what you expect? It is of no use having a technically correct program that does the wrong thing! This is usually the case of understanding the requirements or quite often things like testing for "greater than" when it should have been "equal or greater than" These problems can usually only be found by visual inspection (code review) and thorough white box testing. For catching errors during code inspections, the code needs to be readable. This is something I will return to later.

Repeatability is one thing that is often overlooked when testing software. Most software (and embedded in particular) often has to perform the same tasks many times, sometimes for years on end. I have used a program that ran well for a while (3 months). It then crashed but after a reset, it ran again (for about 3 months). It was, under some situations, over-writing buffers. Unlike desktop PC's embedded systems have to be reliable as they do not have a ctrl-alt-del. On one project, I worked on the life of the system as 15 years. That is it had to run continuously 24/7 for 15 years. They were designed to run for 20 years just to be on the safe side.

Also, embedded systems often control machinery. Malfunctions in robots making cars in Japan killed 6 people in one year! However, some of this was due to EMI problems and not the SW as such.

The last point on the list is that if you can't *easily* read the code and it is not written to be clearly readable you will not be able to check for errors and the correct running of the code. There are two parts to this firstly the code should be using the correct constructs in a safe manner. The best place to start is with the international standards and then the industry specific ones. In this case, the international standard is ISO C. (Not ANSI which is a local USA standard). A de-facto standard for c usage is the book "C Traps and Pitfalls" by A Koenig. More recently MISRA-C has gained a lot of standing and is used across the embedded field but I will look at the embedded parts of C later. Secondly, to be easily readable the source needs to be uniform in appearance.

You may well (or at least I hope so) agree with the first part of the last paragraph but may disagree with the last line. Many people do not like to be told how to layout their code. Many see it as an infringement of their civil liberties. However, sw *engineering* is a branch of engineering not a mystical science!

As a final though o "good C" I give you this quote from Brian Kernighan :
Debugging is at least twice as hard as programming. If your code is as clever as you can possibly make it, then by definition you're not smart enough to debug it.

4.1. Style

Style is often the more contentious area to get people to agree on as it has no mechanical bearing on safety. I have a few examples that I hope will show you that having a uniform style (for the whole project) is the best thing to do.

It is easier to count a group of people if they are standing in lines of 5 and blocks of 25 than if they are just standing in a group. This is why the army make troops stand in lines. To prove this for yourself tip a box of paper clips on to your desk. Without touching them, count them. It is not that simple and it is easy to make mistakes. Put the paper clips in rows of five and blocks of five rows. Now it is possible to count the number of paper clips at a glance. Count them again…. If they are in a heap it will take the same time as it did the first time. If they are still in the rows it takes a fraction of a second.

Likewise when the source code is laid out is a standard way is it far easier to spot anomalies and errors. It will not take any longer to write it to a particular house style but the time saved every time you have to read it will soon mount up. Also as with rows of paper clips that stand out because they only have four clips in them errors in the source code will be easier to spot.

As a final proof that a standard layout of source code will save time and reduce errors is email sent to me by one of my team after doing a review on another teams code:- (N.B. time how long it takes to read what this says)

```
th  eo          t
her     tea     mRe                    Gua
R       d           so     ru  c
E           Co                     DeLay
O           T       A                  Sana
Rtf     or                 M
```

It took me a while to work out what it actually said was:-
ThEoThErTeAmReGuArDsOrUcEc0DeLaYoTaSaNaRtFoRm

Sorry, I meant "**theo tert eamr egua rdso ruce code layo tasa nart form**" or, according to some free thinking software engineers, to restrict my civil liberties and stifle my creative spirit; "*The other team regard source code layout as an art form*". I am sure that you instantly spotted the '0' (zero) in place of the O and the misspellings. In fact now I come to look at it, the first 3 versions have different errors but I am sure that was obvious to you!

The illustration above should have convinced you that a uniform style to a set convention is a good idea. If only when it comes to saving time looking for the bugs. Don't say, "what bugs?" Zero Defect software is reputed to be a myth as so few have achieved it. Those that have usually use very strict style guides.

There are many style guides about. Have a look on the internet or create your own. For those who want a ready made guide I have produced one as part of the **QuEST** series called the **Tile Hill Embedded C Style Guide** it is also available at http://quest.phaedsys.org/ .

No matter which style you use do so consistently.

There should be a consistent style on a project not just per engineer. A consistent style across the whole department or company is better.

NOTE:- (and this is important) Style is about *readability*. It is easier to spot mistakes if something is easy to read. Style guides are not about safe code as such or safe subsets of C. Safe subsets of C we will come onto later.

We now have a religious style debate, which has caused more lines of emails and messages to newsgroups than lines of code in the programs referred to! This topic, more heated than any discussion on faith is about "where to put the braces". There is no “One True Faith”. The truth is:- Any *system* will do as long as you stick to it! Some of the more common are styles are shown here.

4.1.1. (K&R)

```
If(xyz){
     statement
     statement
}
```

This is the original style from 30 years ago. Because it is the "Original" it is perceived by many to be "The Way". Some of the (very) old tools look for this pattern of the opening brace on the same line as the if. Most modern (ie windows) tools are happy to accept any style now.

4.1.2. (Indented)

```
if(xyz)
     {
     statement
     statement
     }
```

This is a common style that came in after K&R when people realised that more lines on screen did not take up more compiled space. Also as screens could show more than 25 lines (remember those?) and were able to show 50- 60 or more code did not need to be so cramped.

4.1.3. (Exdented)

```
if(xyz)
{
     statement
     statement
}
```

Exdented, (or outdented as the Americans call it) my preferred favourite has the advantage that the braces are easy to spot. And visually link into pairs. (and easier to make up in pairs with a pencil when printed out.

I have come across some other very strange methods but I would suggest that the common ones are common because they have been found to work over the years.

A fuller description of each is given in appendix A. Personally, the extended is my preferred style but some of the older debugging tools require the first style. A full style guide, ***The Tile Hill embedded C Style Guide*** is now available at quest.phaedsys.org As with this paper any comments are welcome.

There is one place where style and safety meet. The one thing I insist upon for braces is that they are used wherever they can be used. This is especially important on things like if clauses for example

```
interlock = OFF;

if(TRUE == stop)
     flag = ON;
     interlock = ON;

if(ON == interlock)
     open_doors();
else
     apply_breaks();
     sound_alarm();
```

This will, obviously, always open the doors and sound the alarm but not apply the breaks! What it should do is only open the doors if stopped else apply breaks and sound alarm but not open doors.

I insisted that all code produced with teams I am involved with rigidly adhere to the principal of always using braces were possible. This may sound a bit draconian but I have good reason.

I instigated this rule after three of the team spent two days trying to trace a bug caused by a two line if statement where only one line was actually inside the if. It caused an error some distance from the if statement and was not immediately linked to the problem. When the if statement was considered all three engineers glancing at it saw a correct if statement and mentally put braces round the two statements. The mind saw what it thought should be there. The error was combined with another similar "non error" to produce a real problem much further away.

The previous code example (according to my pedantic formatting) is actually the following:

```
interlock = OFF;

if(TRUE == stop)
{
     flag = ON;
}
interlock = ON;

if(ON == interlock)
{
     open_doors();
}
else
{
     apply_breaks();
}
sound_alarm();
```

Whereas what was meant was:

```
interlock = OFF;

if(TRUE == stop)
{
     flag = ON;
     interlock = ON;
}

if(ON == interlock)
{
     open_doors();
}
else
{
     apply_breaks();
     sound_alarm();
}
```

This is actually based on a real problem on a rapid transit system in the far east….. written by programmers in the Midlands! It actually made it as far as the test runs. The problem was only found by accident after a carriage broke down and the test train ran with one fewer carriages than normal.

The fixing of this fault cost £50,000 in time, phone calls, faxes, and an Engineer who went out to investigate.

The code was later given to a Static analysis tool vendor to see if they could spot the problems that caused the bug (The actual cause that showed the problem was the reuse of a variable somewhere else).

The static analysis tool that (at the time) cost £5,000 and £2000 to set up and configure for the project found the original cause and the problems with the if statements in 7 minutes. Interestingly I believe it found a few other "anomalies" that they did not know about. The company bought the tool and spent a frantic few weeks ironing out a few things and getting an "upgrade" out to the customer…. I refer you to the section in the introduction that discusses corporate manslaughter!

Static analysis tools start at £127 for PC-Lint (including the configuration) so you have no reason not to use static analysis. PC-Lint would have picked up the errors in the case above.

4.1.4. Information blocks and comments

Comments, or the lack of, are one of the most hotly argued things after where to put the braces! A full description of commenting is in the style guide called the **Tile Hill Embedded C Style Guide** as part of the **QuEST** series . It is also available at quest.phaedsys.org.

Each file or module should have an information block similar to the one below.

```
/*******************************************************
** $Workfile:   U1CO0001.C  $
** Name: Application Block
** Copyright :PhaedruS SystemS 1999
** $Author: Chris Hills$
** $Revision:   1.1  $
**
** Analysis reference:123/ab/45678/001 5.6
**
** Input Parameters:  NONE
** Output Parameters: NONE
**
** $Log:   C:/ENG/KOS2/A2C001.C_V  $
**
**   Rev 1.1   06 May 1998 16:48:28   HILLS_CA
**Issued for review
**
**   Rev 1.0   01 Apr 1998 13:09:02   HILLS_CA
**initial version
**
*/
```

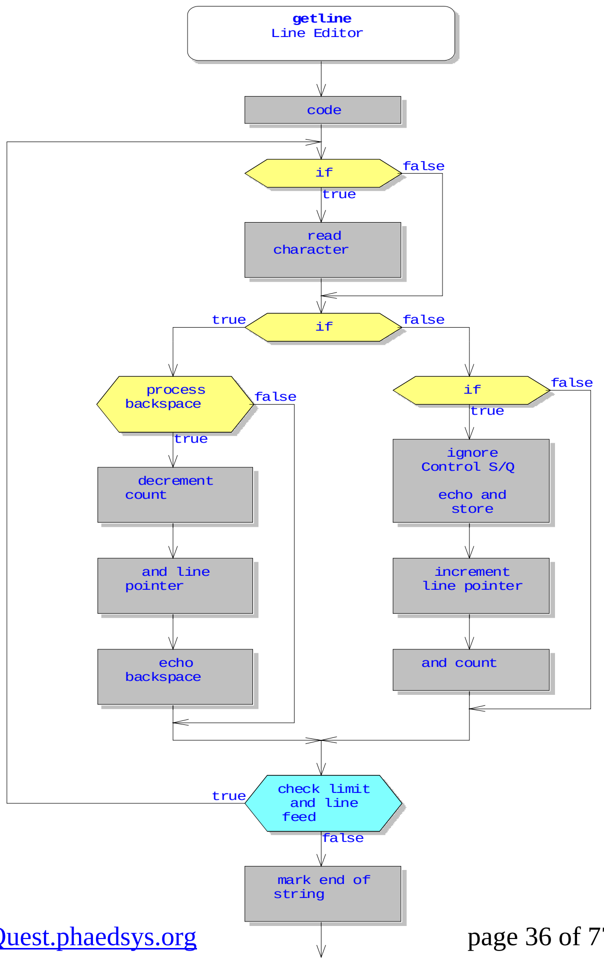
This is an example that was used in a project under ISO9000. Some of the significant points are the Analysis Reference to tie the source to the design documentation and the history log. This should give the developer all the information on where the file started and how it got to its current state. The history block in this case is automatically put in by the VCS. Where a VCS is not used it should be manually maintained.

Each function should also have a simple comment block giving the purpose of the function, the input and output parameters. Like the main file information block, where appropriate, the reference to the design or requirements should be included. This may sound like a lot of extra work but I have found that it makes one focus on why the function is there.

```
/************************************************************************************ Convert_One */
/* Name: Convert_one
**
** Purpose: Converts Faranhit to Celsius
**
** Input Parameters
** Return Parameter
**
*/
```

The complexity of the function block can be adjusted from a standard template to suit the function. A simple function to add two numbers and return the answer will need fewer comments than a function that manipulates several parameters and outputs to (or reads from) a peripheral.

It is a moot point if the function comment block should or should not contain all the information on the algorithms etc used in it. This will depend on how good your documentation is. Hopefully a reference to the design document is all that is required. The only time I put (almost) as much comment in the code as code was when the source had to be self-documenting as it was going into the public domain without any other supporting documentation. Beware commenting the obvious. Too many comments can be worse than too few in some (rare) cases.



The one thing I do find useful is at the start of each function to have a single line comment full width on the page with the function name at the right hand end. This makes finding functions easy when scanning listings.

The biggest controversy is where to put comments in the source (if at all). Someone asked the question "where should I put comments in a program" recently (August 99) on one of the C news groups on Usenet. The thread attracted an order of magnitude more replies than any other (except the one whether C++ was suitable for embedded use). I scanned the thread but there was no clear winner or consensus. The suggestions went from commenting every line to the idea that well laid out code needs no comments at all. The answer is somewhere between the two. I use DAC, see diagram opposite.

This can take source code and automatically produce metrics and flow charts. The structure of the flow chart comes from the source, the analyser reads the if, switch, do while clauses etc. However, the program takes the comments in the code to put in the boxes of the flow chart. I use this program as a guide to where I need comments. The rule of thumb is that comments should be 30% of the file. This figure is only a guide. Remember the golden rule:-

Comments should be used to make the source readable by *another person* not the original developer.

It has been pointed out to me that, if given some time between readings, the same programmer IS another person. One Engineer (Jonathan Kirwan) said to me "I've gone back over old code I've written and wondered what I did -- even WITH comments there. They just weren't the right ones!"

The only strict rule is that comments should not be nested. Compilers and other tools are not guaranteed to handle it in the way you might think.

```
/* for example

/* this nest comment
ends here*/

some might see this as an error */
```

This is because they go from "/*" to the first "*/" There is no guarantee that the compiler or tool will count the opening "/*" and match the last closing "*/"

As an extension of this rule, code should not be commented out because it could contain comment blocks. Also commented out code is confusing and can be, due to the way C nests comments, not the commented out block the developer thought it was.

Originally I wrote All comments should use the /* */ pair not the c++ style of //. This is because not all C compilers support the C++ style even if BCPL did! Now most C compilers do support the // commenting style. I would relax this rule if you know your code is only going to be used on a tool chain that supports it. Most embedded code is not portable across many platforms. In fact it is usually written very tightly to support a particular target.

For code that is intended to be portable I would still be inclined not to used the // comment system.

4.2. Header files.

Header files contain all sorts of things such as defines, macros, and function prototypes that are required in several files. The standard libraries have header files that are usually included. These are included into the source using the #include directive. This is a straight textual insert. What can go wrong? Lots of things….

Since writing the second version of this document I discovered, on a customer's site, a major thing that can go wrong…. Not having header files in the first place!

Why do you need header files? Because header files minimise errors. Whilst re-writing this paper over Christmas 2003 I cam across this in the mail list acc-general (see www.accu.org) It was from an experienced software developer of many years standing:-

```
Anyone remember life before prototypes?  How unpleasant. My
first real C program, I wasn't very hip to methodology yet.  I
didn't know about either incremental development or up-front
design.  What I did was write a 10,000 line ray tracer (for
lens design, not graphics) in C before I tried to compile it
the first time.  It took a couple weeks to get it to compile,
and when it didn't run I abandoned it, as I had no hope of
debugging it.

That's when I discovered header files for the first time,
because I got tired of repeating structure declarations in
different source files.  I knew I had to put #include <stdio.h>
at the top of all my source files, but I had no idea what it
was really for.
```

Header or include files are very powerful and flexible. You will notice that in C there are two types of include line:-

```
#include < stdio.h>
```

and

```
#include "myheader.h"
```

The file in the <> brackets indicates to the compiler that this is a system file (i.e. the libraries that come with the compiler) and that the compiler should first look in the compiler include directory. Note that many specialist libraries will also be loaded into the compiler library. If your company develops it's own libraries (these must be fully debugged and documented) these too should also be installed to the main compiler library area. It is common practice to place these in a sub-directory for example:

C:\compiler\lib\nag

And

C:\compiler\inc\nag

These would be called as follows:-

```
    #include < ./nag/math.h>
```

This is the only permitted use of paths in an include directory.

Where the file in the "" indicates that the project directory is the first place the compiler should look. This is a pretty large hint that the originators of C thought that you would be creating your own header files.

There are many reasons why would you want to create your own header files. To take a simple example think of the stdio.h file. This includes the function prototypes for:-

```
extern char _getkey (void);
extern char getchar (void);
extern char ungetchar (char);
extern char putchar (char);
extern int printf   (const char *, ...);
extern int sprintf  (char *, const char *, ...);
extern int vprintf  (const char *, char *);
extern int vsprintf (char *, const char *, char *);
extern char *gets (char *, int n);
extern int scanf (const char *, ...);
extern int sscanf (char *, const char *, ...);
extern int puts (const char *);
```

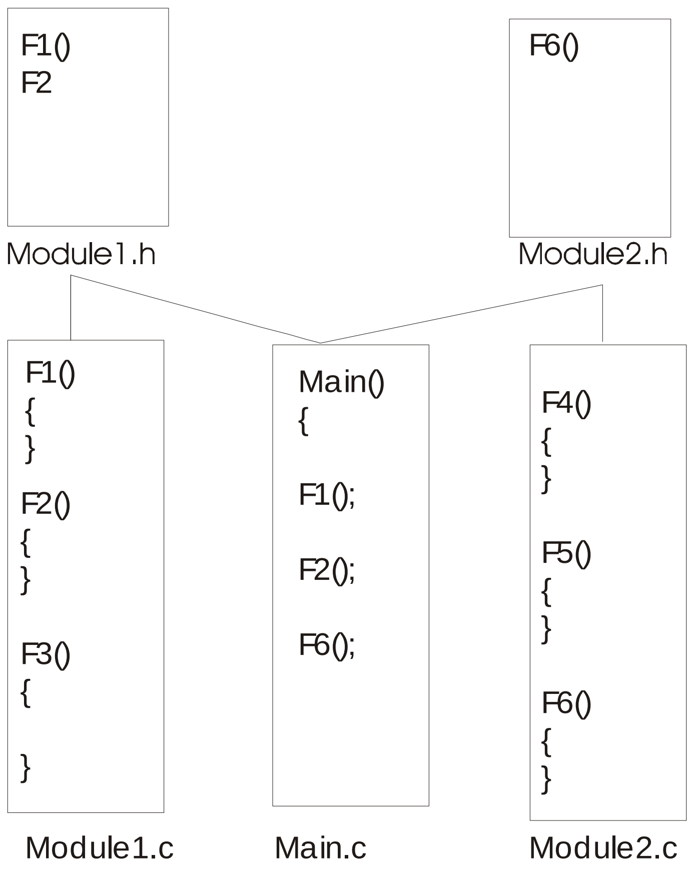
When you include sdtio.h it means you don't need to put that lot into every file in which you want to use the io functions. In addition, the header file often contains information on usage. However, with the standard header files the information is usually in the compiler documentation.

Your own header files will work in the same way. Where you have functions, defines and macros in a source file that will be used outside that file, they belong in a header file. Note:- **a header file should never contain executable code or declare storage space.**

I have seen a case where several C source files were #included into the main file. This is extremely bad practice. Added to which most debuggers, simulators and ICE cannot cope with this. As soon as you start to run any high-level language debugging the system will crash.

Functions used only within the source file should be declared as static and not put in the header file. See section on static linkage.

In the diagram shown "module1.c" has 3 functions. Two of these functions, F1 and F2 are used outside the file. These are therefore put in module1.h. Likewise in module2.c the function F6 is used outside the file so it should be placed in module2.h Note ***all*** files that are intended for use outside should be in the header.



In this case, the files module1.h and module2.h would be included in main.c

Because Module1.c does not use any functions from module2.c the header file moduel2.h would not be included in module1.c.

I must stress that each file containing functions that are used externally needs its own header file. You should not have only one header file for all the external functions.

The reason for this is to do with encapsulation, data hiding and linking. By making the internal functions static they cannot be seen outside the scope where they are declared (block, function or file) and often use faster smaller jumps. By only including the header files a source module actually needs it can only see the functions, defines, macros etc it needs. This helps minimises any errors and helps maintenance.

Should for example the prototype for F6 change you will have to change it in Module2.c and module2.h, this will automatically change the "extern F6();" in every file where the function is used. All you then have to do is change the actual calls. A static analysis tool like PC lint will show up the uses.

The alternative, without using header files, is to change all the "extern F6" lines in all the modules where it is used. The problem here is that quite often, programmers being lazy, all they do is cut and paste the new version in. You then get lots of unused prototypes cluttering up the place and causing a nightmare in the future. The classic "update problem" applies, of course. "Information" should appear exactly once, somewhere.[5]

[5] .Note from Jonathan Kirwan: In fact, the methods documented by C.J. Date on relational database design are excellent and can be applied on how to craft modules

The other useful point about header files is that, as with the standard library, it gives the key to the interface to the module. The *.c file can be compiled to object code and all the information the programmers need to use the module should be in the header file. You must remember to update the information in the header file should the usage of the functions change.

<u>Header files should not be nested.</u>

Nesting hides files. On one project I worked on in one file there were 8 include files. However, these files had nested files. When I unravelled, the nested headers there were over 120 included files. This included a set of 8 files ten times! The interesting point was that when the duplicates were removed the source file would not compile! It appeared that due to some problem no one could be bothered to find, three of the files had to be included twice. Once at the start of the #includes and once at the end. This was masking a serious problem.

For example something may be defined in one header, turned off or redefined in another and as the two get repeatedly included parts of the overall included files will have the define in and others not or the valued of the define changed. This can have some very strange, and almost impossible to find effects.

In order to make sure only one of each header is ever included guards should be used. These guards take the form:

```
#ifndef name
#define name

header file contents

#endif
```

Generally for *name* I use the name of the header file. For example "_STDIO_H" or "_MODULE1_H"

An additional tip is to always include header files in the same order. I usually start with system headers at the top, followed by any general project files then the rest of the files. Some people insist that they include files in a set order, or alphabetically etc. Any system is better than none. It pays off in the long run… usually when bug hunting at 16:30 on a Friday with a Monday morning delivery.

4.3. Macros

Macros should only really be used for constants and function like macros. Using a macro for something like :

```
#define STARTIF  IF(
```

should be avoided at all costs. In all cases they should start and end with ( ) unless the are a single item.

Incidentally when putting macros into headers use parenthesis enthusiastically to ensure there are no silly side effects. I.e. the macro is self-contained.

```
#define MULTIPLY(a, b)            ((a) * (b))
```

rather than

```
#define MULTIPLY(a, b)            (a * b)
```

Observe what happens if someone calls MULTIPLY(my_value + 1, old_value). The second form yields the sum, rather than the product

4.4. Magic Numbers

Defines & magic numbers. I hope I do not need to tell people not to use magic numbers, #defines or const should be used. The advantage of using const is that it will be visible in debuggers. The disadvantage is that it will actually take up space in ROM or RAM as a (const) variable.

So whilst const is preferable technically, pragmatically #defines may be better in 8 bit systems like the 8051 where RAM is at a premium. As an example of using defines read the following code.

```
case 0x01:                         // Reset Machine
    transmit_handler(0x10);
    reset_machine();
    if (timed_out)
    {
        transmit_handler(0x60);
    }
    else
        transmit_handler(0x20);
     break;
```

With this code one has to guess at what is going on. The reader has no idea what case 0x01 is. Or how it is related to any other of the hex values in the code. Now see the same code with the magic numbers changed to defines.

```
case  RESET_MACHINE:
         transmit_handler(BUSY);
         reset_machine();

         if (TRUE == timed_out)
         {
```

```
                        transmit_handler(TIMED_OUT);
                }
                else
                {
                         transmit_handler(READY_RESET);
                }
                break;
```

The second version is readable and less prone to errors. If the define is wrong it is more likely to throw up symptoms across the whole program. In many cases the use of defines (along with meaningful variable and function names) can go half way to documenting the source.

4.5. Flow Control

Controlling the flow of a c program has always caused problems. People always tend to take the shortest route. It has already been mentioned that IF clauses should always use the braces {} even when there is only a single statement. This holds true for while statements as well:

```
while(a<count)
     a++ ;
```

is potentially dangerous. It should always be written:

```
     while (a<count)
     {
          a++;
     }
```

4.6. If, while etc

Something I have found effective in tests for equality if to have the fixed or constant value on the left and the variable on the right. This is the opposite of the common way of writing it eg

```
        If(variable == constant)
```

This causes an error if, inadvertently, the test for equality is inadvertently changed to an assignment. For many years, I have written

```
     if(constant == variable)
```

This is counter intuitive and does take a while to get used to. However, it does stop many silly errors. Though with a good compiler and rigorous use of lint any errors of this type should be picked up whichever way it is done.

Logic is one of the problems when using if with else. Where if else if is used there must always be a final else clause. This should be done even when the final clause will be empty! A comment should be placed in the final else to say why it is empty.

```
If( Clause)
{
     statement;
}
else if(clause)
{
     statement;
}
else
{
     statement;
     /* or comment*/
}
```

This makes it clear why the clause is empty and makes the developer think about the structure of the whole construct.

Also where there are two mutually exclusive ifs such as

```
If(True)
{

}


if(False)
{

}
```

should be written as

```
if(true)
{

}
else
{

}
```

If it is not true it must be false? This may not be true where True >=1 and False == 0 . What happens if somehow the value is -1 Don't assume that the value will only be the ones you expect.

4.7. Switch

Switch statements are completely straightforward, what can go wrong? Lots of things can go wrong! All switch statements must have a default clause. If there is no default action put an error message in there. This saved a lot of grief more than once when (completely unexpectedly) the error message showed up! Break should be used to terminate each case.

```
Switch(variable)
{
        case 1: statement;
                break;

        case 2: statement;
                break;

        case  3:
        case  4: statement;
                break;

        default:
                printf("ERROR!!!\n");
                break;
}
```

Notice the break on the default. Always a good idea just in case the default becomes a case.

4.8. Breaking the flow

In a word don't. Break should only be used at the end of every case and default clause in a switch statement and nowhere else.

Goto….. This needs no comment, as it should never be used. Neither should continue.

Whilst on the subject of jump out of a flow it is a moot point whether there should be a single point of exit from a function. Many say there should only be one return. Others say that for readability and sensible flow in a function more than one is better. I have seen cases where the function contained some horribly complex if else if, constructs in order to get one return line whereas it was far cleaner, elegant, shorter and faster with several return statements.

4.9. Static linkage

By implication, all functions are externs. However, where a function is only called within the module it is in it can be made static. This has a couple of uses.

Firstly, it makes the code safer in that visibility can only be in the module. This means that the function can only be called in the module. When used with static variables declared at file level it makes the variable public to only that file. This is rather like the private functions in a c++ class.

Secondly, static functions can result in faster code. This is because the compiler knows where the function can be called. Generally, this will be a local, short or relative jump within the file. This will be tighter and faster than a general jump to "somewhere else" out of the file. For embedded use this has the advantage in saving a few bytes per call.

4.10. Declaration and initialization

Variables should be initialised before they are used. In keeping with the general rule of explicit rather than implicit variables should be explicitly initialised as soon as possible. The obvious and most sensible time is when they are declared.

```
int count = 0;
signed char  letter ='a';
```

This ensures that all variables are initialised: Variables should always be declared and initialised at the start of a file or function. Automatic variables, which are local to a function, are created as required but are not automatically initialised and contain random data.

Static variables (see ISO C 6.2.4.3) are initialised to 0 at the start of a program (before it gets to main) and in the case of embedded systems will be placed in global memory. So the declaration:-

```
static int count ;
static signed char  letter;
```

will ensure that both are at zero on first use. Of course zero may not be what you want if it is a character.

5. How to prove it is Good C

When "The Gods" [K&R] gave the world C and the world bathed in the brilliance of the language it rather overshadowed another program that "The Gods" had left for the disciples. Those who know it use it religiously. Many, to their cost, leave it by the wayside. What is this program? Lint. It was first developed from a C compiler engine in 1979 by Steve Johnson [Johnson] who was one of the original group that worked on C and UNIX.

In his paper on C Ritchie [Ritchie] says- "To encourage people to pay more attention to the official language rules, to detect legal but suspicious constructs, and to help find interface mismatches undetectable with simple mechanisms for separate compilation, Steve Johnson adapted his Pcc compiler to produce lint."

Whilst a compiler will very accurately check *syntax* it does not worry about the *semantics* as long as they are legal. Now legal does not mean sensible or safe. A compiler may not care if you want to store part of an int or float into a char. The fact that it makes no functional sense is irrelevant to the compiler!

To give an example from English language, in a meeting I attended a person (who's first language was not English) said "I have over seen that." What he meant was "I have over looked that." The meaning he gave was I have personally checked that whereas what he meant was "I forgot about it." Lint will do much the same sort of thing for C.

Going back to the dubious if statements used to illustrate the need for braces:

```
1       interlock = OFF;
2
3       if(TRUE == stop)
4               flag = ON;
5               interlock = ON;
6
7       if(ON == interlock)
8               open_doors();
9       else
10              apply_breaks();
11              sound_alarm();
```

If lint was run over this code it would complain that lines 5 and 11 had incorrect indentation. In fact on PC-Lint it complained:

```
"Warning 539: Did not expect positive indentation from line 5"
"Warning 539: Did not expect positive indentation from line 11"
```

If you refer to the PC-Lint manual it gives an if statement without braces as the example!

Another subtler problem on the same lines, from the lint manual, is:

```
if(….)
        if(….)
                statement
else
        statement
```

The else is in fact part of the second if, not the first. This is where a good style guide is useful so the code is uniform and insisting that braces be used on ***ALL if, do and while clauses.***

Since lint was developed, there have been great strides in static analysis. This is where the analysis of the source code without compiling it or running it. HP has estimated that static analysis and code inspections are 5 times more efficient than white or black box testing. Alcatel have said that static analysis can reduce a project time by up to 30%, primarily from the debugging and fixing stages. The cost of fixing a source code error (other than syntax) rises exponentially the longer it it left and the further down the development it goes. Thus the most cost effective way of fixing sw errors is at source when the engineer is writing the code.

I use the "write and lint" cycle instead of the more common "write and compile" cycle to check the code.

Lint is not the only tool. It was the first one for C and in keeping with its Unix/C roots is a simple command line program. Other heavyweight static analysers (that are also vastly more expensive and time consuming to set up) are also available. See http://www.ldra.com/ and http://www.programmingresearch.co,uk/

Apart from using lint ***always run the compiler on it's highest level of error checking***.

MISRA-C is a very good set of rules for using C in safety-critical embedded situations. PC-Lint has a configuration file to test for as many of the MISRA-C rules as is possible statically. MISRA-C also shows how to construct a conformance chart. I would recommend this to any developer who needs to show "due diligence" or prove that the system has been tested.

5.1. Formal Eastern European Writing

There cannot be a discussion on safe C, "proper C" without someone bringing in the two methods for producing perfect code. These methods in *theory* are very good. Theoreticians usually put them forward. Their practical use in real sw engineering is another matter.

Both the methods should, in theory, eradicate many errors and mistakes but, in my view, create more than they solve.

5.2. Hungarian Notation

This is a method were the name of a variable conveys information as to the usage and the type of the variable. This method sounds wonderful until a type is changed part way through development. It also does not help readability. There are also several standard notations in use and countless local ones. This breeds confusion.

Shown below is an example of Hungarian notation from Steve McConnell's [McConnell] book Code Complete.

```
For(ipavariable = paFirstvariable; ipavariable <= paLastvariable; ipavariable)
{

}
```

further examples:

ch	a variable containing a character
ach	array of ch
ich	index to an array of ch
ichMin	indest to first character in array
ppach	pointer to pointer to array of ch

```
mhscrmenu
 m      module level
   h    handle
     scr   to a screen region
        for a menu
```

This is logical BUT I have found that these methods usually cause far more trouble they are worth. There appears to be no general standard. After you learn one, someone changes it. I have seen a project where it was 2 weeks before the team discovered that some of the team, whilst using the same letters were using them to signify different things from the rest of the team!

5.3. Formal Methods

Formal methods are a mathematical way of describing a program. They are NOT a programming language though there are some interpreters for at least two of the languages (Z and VDM)

The problem is that there are two interfaces. One takes the specification and turns it into Z or VDM and at the other side the conversion from the formal method to the programming language of your choice.

The interesting thing is that due to the absolute certainty of people in these methods it can cause problems. Folk history has it that a well-known CPU vendor used formal methods in its chip design, for the microcode. When several thousands of these chips were produced (at a cost that could bankrupt many companies even now) they were shocked to discover a bug!!!

It transpired that an error was introduced in the translation between the requirements and the formal methods. The formal methods were OK as was the translation to silicon. The problem is that it is generally the mathematicians who like the formal methods.

What do formal methods look like? This is VDM

Max(s:X-set)r:X
Pre $s \neq \{\}$
Post $r \in s \cap \nabla j \in s.r \leq j$

This is a function to find the largest element in a set. I have managed most of it as even with the might of Windows 2K and Word 2K I do not have all the correct symbols!

As I said: wonderful in theory (especially among mathematicians) but a nightmare in practice with software engineers.

6. Embedded Engineering

Embedded Engineering, whilst having many similarities with “ordinary” SW Engineering, is different. Different, that is, from “ordinary” software Engineering and every other embedded project. Whilst embedded systems share many similar attributes no two are the same.

In general, embedded systems tend to be a single task, with interrupts, albeit that the larger systems may have an operating system and many processes running. They usually have to meet deadlines that are far tighter than general-purpose systems. This is because they often have to react to inputs that are measured in microseconds not seconds or minutes.

Another difference that is crucial is that embedded systems are usually built to a minimum cost with little or no room or even facility for any expansion. That is unlike the desktop PC. The resources such as memory are cut to the minimum required for the job and sometimes dictated by reasons other than engineering. This is because, usually, the embedded system is a small part of a larger system. The control system is ancillary to the main function of the system for example a microwave cooker. If additional resources are added the cost of the product goes up or the profit goes down.

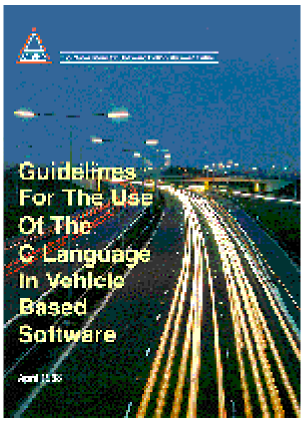
The problem, for the programmer, is that often memory saving techniques have to be used. However the most dangerous problem is memory leaks. I once worked on a comms system where a series of line connections caused a byte of memory to be lost. One of the engineers did some calculations and worked out that the unit would fail in 2 to 4 years of use. The other problem was that depending on usage that the unit had, the failure symptoms could be wildly different. The unit had a lifetime of 10 years and many units would be in remote sites.

There is one other very important aspect of embedded systems: they tend to be used with mechanical equipment that moves. Whilst not all embedded systems are safety critical many are. The others will cause problems if they fail that are usually more of a problem than having to reboot a PC

7. Embedded SW Engineering with C

Now we have reached the significant part. However, this section will be surprisingly shorter than many people expect. I hope that it will only be a surprise to those who have skipped the previous sections. Good SW Engineering practice and good Embedded Engineering Practice should result in the correct approach for Embedded C! There only a few additional tweaks needed for embedded C.

Just as there are good style guides for C there are useful guides for a ***safe subset*** of C for embedded use. Note: This is NOT a style guide.



The de-facto standard is MISRA-C [MISRA]. This was originally developed for the automotive industry. This industry uses 8, 16 and 32 bit processors from many families. Thus, the guide is suitable for virtually any embedded system which is why it has gained widespread acceptance across the embedded world. Coupled with a good style guide MISRA-C will help you produce robust and safe C. PC-Lint now has a configuration file to test for many of the rules (it is not possible to test for all 127 rules statically.

A style guide, MISRA-C and PC-Lint between them should let you produce embedded C code that is safe, robust and readable. What more could you ask for? You will find most of the points covered in the following section in MISRA-C and the PC-Lint Manual.

7.1. SIZE MATTERS !

As stated previously embedded systems tend to be of fixed size for the lifetime of the item. In addition, memory and resources cost money. Memory tends to be the most expensive component in an embedded system. Designing in the additional chip, tracking the larger board that will cost more to make all add to the costs. In many cases a single chip MCU is required (due to space and costs) therefore the memory available will be physically fixed. Space is expensive and restricted so make good use of it.

I know of one project where there was a white board where the memory available was displayed down to the last bit! (This was a specialist system where only a specialised purpose built MCU could be used.) There was actually bartering between the developers for the memory.

Memory usage can be decreased by several ways.

Use appropriate sizes of data types. This may sound obvious but it is surprising how many people make incorrect assumptions. I have been told that short is more portable than an int and that a short is always 8 bits. Neither statement is correct.

Objects may be non-nutritive sizes; Pointers are not always the same size as ints, the same size as each other, or freely interconvertible. The following table shows bit sizes for basic types in C for various machines and compilers.

type	pdp11 series	vax	68000 family	Cray-2	Unisys 1100	Harris H800	80386
char	8	8	8	8	9	8	8
short	16	8/16	8/16	64(32)	18	24	8/16
int	16	16/32	16/32	64(32)	36	24	16/32
long	32	32	32	64	36	48	32
char *	16	32	32	64	72	24	16/32/48
int *	16	32	32	64(24)	72	24	16/32/48
int (*)	16	32	32	64	576	24	16/32/48

Type	Minimum # Bits	No Smaller Than
char	8	
short	16	char
int	16	short
long	32	int
float	24	
double	38	float
any *	14	
char *	15	any *
void *	15	any *

Some machines have more than one possible size for a given type. The size you get can depend both on the compiler and on various compile-time flags. The following table shows ``safe" type sizes on the majority of systems. Unsigned numbers are the same bit size as signed numbers.

Typedefs should be used for the size of data the types. Furthermore, all types should be explicit not implicit. By which I mean that there should not be a "char" but either an unsigned char or a signed char. Do you know, off the top of you head, if your compiler defaults to signed or unsigned char?

I use the typedefs shown below. Note I have not used "int" in any of the names. This is because much of the time it is a container for data rather than an integer.

```
typedef unsigned char       uint8_t;
typedef signed   char       int8_t;

typedef unsigned int        uint16_t;
typedef signed     intint16_t;

typedef unsigned long       uint32_t;
typedef signed   long       int32_t;
```

These typedefs are placed in a short universal header that is included in all files. Short should only be used where it is larger than a char and smaller than an int. For example:

Char	= 8 bits
Short	= 16 bits
Int	= 32 bits
Long	= 64 bit

The use of lint with strong type checking enabled will cause warnings to be issued where one type is assigned to another even if the underlying type for both is an int.

7.1.1. Integer promotion

Integer promotion is a problem that is not really understood by many.

7.1.2. Enumerated Types

Another place where space is wasted is in enumerated types. The ISO standard says that an enum is 16 bits. This can waste a lot of space on 8 bit systems when the enum only has a few values. There are several ways round this. Some 8 bit compilers will use a char to hold enums where possible. Another way is to use #defined values. This can also speed things up on 8 bit systems as the define and the 8 bit enum both fit in a char. Manipulating chars is much faster than 16 bit data on an 8 bit system.

The use of lint can improve memory use. When used across all the files in a project lint can pick up globals that can be declared as locals, variables public to a module that can be local and unused variables.

7.1.3. One way of NOT decreasing the code

```
        if( get( start +offset1 + calc_offset(start+offset2))
        {
                code statements;
        }
```

This will generate almost the same code as

```
temp1  = start + offset1;
temp2  = start + offset2;
temp2  =  calc_offset(temp2);
temp1  =  temp1 +temp2;
temp1 = get(temp1);

if(TRUE == temp1)
{
code statements;
}
```

In this case, one compiler I tested this on the difference was 4 bytes. Whilst I have said that saving even 4 bytes is a good idea in this case it is not. This difference can depend on how the compiler does temporary variables at the point in question. This depends on the compiler, the architecture of the MCU and what else the program is doing prior the point in question.

The difference in source layout is that using a source level debugger or ICE can stop on each line to the second example and look at the temp values. In addition, each stage of the calculation of temp1 can be checked. With the first example, all that happens is that whole line appears to be executed in a single step. One can of course drop in to assembler to step through but that breaks the flow of debugging and the whole point of a source level debugger.

7.2. Volatile

Volatile is a useful keyword for embedded work. Volatile means to a compiler: the value of this could change without the program touching it. This is essential in embedded programming.

I have recently seen a very fast memory test routine. It was so fast because the compiler optimised out a variable, as it was not being changed between accesses. What it did was a memory test without ever touching the memory!

7.3. Const

Const should be used where something is not going to change. One place it is very useful is in function parameter declarations. For example

```
    static uint8_t  func( const uint8_t name, const uint8_t number);
```

This will have both the compiler and lint screaming if the function tries to change the values.

Interestingly enough it is possible to have a const volatile. The application can not change the const but it will change in between accesses without the application touching it.

7.4. Register

Register is often used to speed things up. However, it usually has the opposite effect! This is because the compiler is very good at shunting things in and out of registers and memory. Also it is only advisory, and many compilers ignore it. In those that do not, forcing a variable into a register could give the compiler problems sorting the other variables. In the end, you get a slower system.

7.5. Dynamic memory

The dynamic allocation of memory is dangerous at the best of times as it can lead to memory leaks. In embedded systems, it could be fatal in a very real sense. Much of the dynamic memory allocation routines are not fully specified in the standards and their behaviour (intended or otherwise) is not dependable. I have seen a system where a one-byte leak in unusual circumstances would cause the system to become unstable after 3 years. The system had a 10-year life.

Basically, this touches on a rule in embedded systems, which is to avoid creating a source of errors other than those that the application itself creates for you. Usually the compiler will detect errors in static memory usage. Errors in dynamic memory allocation occur at runtime and can cripple an application for no good reason. The stack, as dynamic memory, is usually unavoidable so you have to live with it. The heap is usually avoidable especially in smaller systems and is specifically forbidden by many standards such as MISRA-C.

7.6. Libraries.

There are three types of library: Your libraries, someone else's library. The third type will become clear in a moment.

Your libraries, it goes without saying, should be constructed to the highest standards and thoroughly tested.

Other libraries are a problem. People assume that they contain "the best" most compact and efficient code. This is not always (or even often?) the case. I have actually seen cases (long ago and far away) where library code simply did not work. Also the implementation you have may not work the way you expect. It may be slower, or just large and inefficient.

Where you have the source to the third party library, you should rigorously lint them, even those supplied with the compiler. Where necessary re-write them. I have come across libraries supplied with compilers in the past that did not survive static analysis! In fact, it produced illegal code under some circumstances which was discovered by a deep flow static analyser lent to the team on evaluation..

7.7. Tuning Libraries

In the case of some libraries, a module may contain a mixed bag of functions. It may well be worth recompiling the libraries leaving out the functions that are not required. A good library has a very fine granularity so you only pull in the actual code you need.

Another thing to look out for is a library with general-purpose functions. For example, the printf function is large and complex. You probably do not need much of the function. It may pay in the end, to write your own print function that only handles the formatting that you need. This will save space, improve speed and you know how they work. It pays to re-do most of the functions that can accept variable numbers of parameters.

Generally a lot of the C library is not often required for most embedded work. If possible get hold of the benchmarks for the libraries: both size and execution speeds. In some cases it is not worth the effort but if you are going to do a lot of work with a particular compiler it pays in the long run. However some specialist compiler vendors do work hard on their libraries.

Floating point libraries can be especially bad. In particular, in the face of pre-emption from interrupts where some interrupt code uses a floating-point calculation. The was a case with one particular 8 bit C compiler, where the initialisation of some flags which controlled the floating point rounding behaviour and other things was un-initialised by the compiler and library code -- meaning that you never knew exactly how it would run in the face of a reset. Fixing it was easy in this case, but only because the source code was available.

7.8. Maths

The comments I have had from many sources regarding the use of maths libraries and floating point can best be summed up as:-

There is almost no reason to ever shift from the integer domain to floating point domain except when programmers are mathematically naive and just can't handle things otherwise.

Maths is one major problem in embedded systems because the maths libraries tend to be large, often slow and do not always behave in the way you expect. There are ways around this. For some things like sin and cos a look up table can be used. Whilst this takes up space this is data not code (highly relevant to an 8051 architecture) it can run much faster than a function that actually calculates the sin or cos.

Another way of speeding things up is to revert to fractions. To get 75% of 100 look at it as ¾ of 100. Take 100 do an integer divide by 4 and an integer multiply by 3. No floats are needed. Though care is needed on over and underflow. Also Integer promotion can come into play here..

Many peripherals such as ADC and DAC units all use integers only.

Another issue in floating point (supplied by Jonathan Kirwan) . Some people don't realize that it doesn't operate like a mathematicians' real number system does. Nor does it operate like integers. It's in a numerical space, all of its own. If a programmer isn't familiar with the numerical implications, they can write bad code. If a mathematician isn't familiar with the numerical limitations of the system, they can specify bad equations to be followed. Someone has to "bridge the gap" when using floating point. If not, it's not uncommon for errors to leak in, hidden for a long time.

For example, in writing a standard deviation routine, a programmer is likely to go read some mathematician's simple equation for it. It may either be the traditional form using a sum of differences or the expanded form using a modified difference of the sum of the values and a sum of the values squared. Either way, what may be too subtle, either for the programmer or the mathematician unfamiliar with computers, is that sums of floating point numbers aren't necessarily exact. If a large number is encountered first, it's possible that very small numbers in the list simply will be "rounded out" of the sum before they get a chance to accumulate to anything. But if the values are sorted smallest to largest first, then the algorithm can be assured that the small values may accumulate into some usable significance before the larger numbers swamp them. This kind of bug can lay hidden for a long time, just waiting for the right data mix to appear.

But who would think it might be needed to pre-sort before summing in some cases? The mathematician? The programmer? Floating point is just plain dangerous and it takes someone with grounding in numerical methods to routinely use floating point with facility. And such people are truly rare.

Incidentally never do comparisons with floating point numbers. Rounding errors can play havoc and give you almost a random generator!

The other taboo is playing with the bit fields inside a float. There is no global standard defined. There may be a de-facto standard but it is not worth the risk.

8. Embedded C++

As C++ (and OO) became the in thing in the "normal" programming world so C++ is becoming sought after in the embedded world. This is to some extent fashion. I have seen many people looking for C++ compilers for the 8051 "because C++ is better than C".

People also want to use C++ compilers when writing C because "C++ is a superset of C" This was true many years ago. However, the two are now distinctly separate languages. Having discussed the matter with members of the UK ISO C and C++ standardisation committees I understand that there are some parts of C and C++ that have the same syntax that mean different things. So, do NOT use a C++ compiler for C.

I believe that C++ is too large for 8 bit systems and indeed many architectures (8051 for example) do not suit the language. C++ is also in its infancy for embedded 16 bit systems. I have found that C++ works in 32 and 64 bit systems. Indeed Embedded C++ (EC++) is being primarily aimed at 32 bit systems and will obviously work on 64 bit systems. I cannot see any real incentive to develop EC++ "backwards" into the smaller CPU.

Of course, compiler vendors would like everyone to go from C to EC++ as this will mean more compiler sales. This leads on to the other problem with C++ that the development tools (and particularly the debug tools) will be very complex and expensive. If you are on a cost conscious project, and who isn't, the buzzwords OO and reuse translate to expensive..

Just as C was a step away from the hardware compared to assembler, C++ is a step further away. EC++ will need far more RAM than its C counterpart. However C++ also creates classes and objects on the fly. The opportunity for memory leaks is very high. EC++ is being developed to counter some of these problems. Things like templates will not be in EC++.

In 1999 I wrote: "I think that in the next two years C++ will be viable for 16 bit systems upwards". This has proved to be correct. I am still not convinced that this is a good idea in terms of speed, memory resources and deterministic response. However, I would not advocate its use in 8 bit systems even if it does become available.

For embedded C++ see:- http://www.caravan.net/ec2plus/ Where you can get the Embedded C++ "standard" as supported by many compiler manufacturers. This was an initiative started in Japan that has spread worldwide.

9. Conclusion

Embedded Engineering is just that: an **Engineering *discipline*.** Like architects and aircraft designers, embedded engineers should use the discipline of proper construction methods and work within the rules. With practice, this will produce robust and safe systems automatically (and quickly).

Once one has got over the learning curve of doing things rigorously, ones mind is free to design with flair. Most of the worlds great buildings were designed using standard bricks or frames made from standard girders to stringent (long and complex) building regulations. Builders and architects who have been shown not to use the correct methods end up in court or no longer able to practice.

Architects do not complain that their freedoms, civil liberties and rights have been infringed with all the rules. They just design fantastic and safe buildings.

As the IEE, BCS, Engineering Council and the government push to raise the status of Engineers in the UK, embedded electronics and software engineers will be expected to come into line with other professions. IE using defined construction methods. This is already happening in Europe and the USA. Not least because of Product Liability causes.

The game is changing and you WILL be judged by its rules whether you want to play or not.

Whilst some industries require certain standards, for the rest it does not take much to use Lint, MISRA-C, a style guide and to use version control (this is required for ISO9000 anyway). The costs for these tools usually repay themselves in the shortening of debugging on the first project. The potential savings are enormous if it saves you having to go to court.

I am assuming that you are, of course, using a good compiler and debugger. It is of little use writing good solid embedded C if you then use a doggy compiler or an intrusive ICE. Tools are a whole new ball game explained in ***Embedded Debuggers*** [Hills]. Getting the language into a robust state is one thing. Having the (appropriate) tools of the same quality to support it is another.

Good SW Engineering practice saves you having to think about much of the trivial time-wasting parts of a project and lets you get on with innovative and safe designs. I recommend that you read MISRA-C and Andy Konig's Traps and Pitfalls from which I stole the title for this paper! .

I shall finish with the last line of the introduction:

It is well worth buying good quality tools.

and add:-

The ART in Embedded Engineering
comes through
Engineering discipline.

10. Appendix A (style)

I do not intend to go through a complete style guide. See the Tile Hill Style Guide for a complete system. These are a few notes to help you decide which of these common styles suit you (if any). Indentation is up to the user as long as it is consistent. That is consistent across a project not per developer.

There are many styles freely available on the Internet. It is not the end of the world if you do not get to use your pet scheme.

10.1. K&R

This is the original style. However, this has largely been superseded, especially with the change to ISO C where the parameter types are specified in the function line not below it.

Some people still use this because it is the "correct" way of doing things and cite K&R (First edition). C is over 20 years old and even the originators, enlightened as they were, have moved on and learnt more. There is nothing wrong with this style (obviously moving the type declarations) but it is not *the* definitive style.

The only word of warning is that *some* debug tools (usually the older ones) assume this style. They require the opening { to be on the same line as the if, do etc.

```
void change_KandR( from, to)
char *to
char *from
{
      do{
            if('a' == *from ){
                  *to ='A';
            }else{
                  *to = *from;
            }
            ++to;
            ++from;
      } while( '\0' != to[-1] );
}
```

10.2. Indented Style

This is a later style than K&R. It requires more space on screen and on paper BUT it takes up no more room when compiled. This more open style crept in when screens gained colour, windows and more than 80 columns by 40 lines.

```
void change_case( char * from,  char *to)
{
        do{
                if(‘a’ == *from )
                        {
                        *to =’A’;
                }
                else
                        {
                        *to = *from;

                        }
                ++to;
                ++from;
                }
        while( ‘\0’ != to[-1] );
}
```

10.3. Exdented Style

```
void change_case( char * from,  char *to)
{
        do
        {
                if(‘a’ == *from )
                {
                        *to =’A’;
                }
                else
                {
                        *to = *from;

                }
                ++to;
                ++from;
        }
        while( ‘\0’ != to[-1] );
}
```

10.4. Tile Hill Embedded C Style Guide

Over the years, I have found I prefer to use Exdented (when nothing else was specified). This is because I find that when doing code reviews it is easier to spot the pairs of braces round a block. I usually join the pairs of braces using coloured pencils.

A full Embedded C Style guide has been developed as the **Tile Hill Embedded C Style Guide** and is available on the authors web site quest.phaedsys.org

This guide is slowly developing wuith feedback from it's users.

11. Appendix B Lint and Example Program

The following program, BADCODE.C, is one of the example programs provided with our evaluation kits. This program has a lot of errors and is intended to demonstrate the error detecting and correcting capabilities of our tools.

Following are listings of the example program, output from the C51 compiler, and output from PC-Lint. The C51 Compiler detects and reports 12 errors and warnings while PC-Lint detects and reports 26 errors and warnings.

As you can see, the quantity and quality of the error messages reported by PC-Lint is greater than that reported by the C compiler.

```
/*----------------------------------------------------------------------------
BADCODE.C

Copyright 1995 KEIL Software, Inc.

This source file is full of errors.  You can use uVision to compile and
correct errors in this file.
----------------------------------------------------------------------------*/

#incldue <stdio.h>

void main (void, void)
{
unsigned i;
long fellow;

fellow = 0;

fer (i = 0; i < 1OOO; i++)
  {
  printf ("I is %u\n", i);

  fellow += i;
  printf ("Fellow = %ld\n, fellow);
  printf ("End of loop\n")
  }
}
```

C51 Output

When compiled with the C51 compiler, the BADCODE program generates the following errors and warnings:

```
MS-DOS C51 COMPILER V5.02
Copyright (c) 1995 KEIL SOFTWARE, INC.  All rights reserved.
*** ERROR 315 IN LINE 10 OF BADCODE.C: unknown #directive 'incldue'
*** ERROR 159 IN LINE 12 OF BADCODE.C: 'typelist': type follows void
*** WARNING 206 IN LINE 19 OF BADCODE.C: 'fer': missing function-prototype
*** ERROR 267 IN LINE 19 OF BADCODE.C: 'fer': requires ANSI-style prototype
*** ERROR 141 IN LINE 19 OF BADCODE.C: syntax error near ';'
*** ERROR 141 IN LINE 19 OF BADCODE.C: syntax error near 'OOO'
*** ERROR 202 IN LINE 19 OF BADCODE.C: 'OOO': undefined identifier
*** ERROR 141 IN LINE 19 OF BADCODE.C: syntax error near ')'
*** WARNING 206 IN LINE 21 OF BADCODE.C: 'printf': missing function-prototype
*** ERROR 103 IN LINE 24 OF BADCODE.C: '<string>': unclosed string
*** ERROR 305 IN LINE 24 OF BADCODE.C: unterminated string/char const
*** ERROR 141 IN LINE 25 OF BADCODE.C: syntax error near 'printf'

C51 COMPILATION COMPLETE.  2 WARNING(S),  10 ERROR(S)
```

PC-Lint Output

When the same code is parsed by PC-Lint, the BADCODE program generates the following errors and warnings:

```
--- Module:   badcode.c
badcode.c  10  Error 16: Unrecognized name
badcode.c  10  Error 10: Expecting end of line
badcode.c  12  Error 66: Bad type
badcode.c  12  Error 66: Bad type
badcode.c  19  Info 718: fer undeclared, assumed to return int
badcode.c  19  Info 746: call to fer not made in the presence of a prototype
badcode.c  19  Error 10: Expecting ','
badcode.c  19  Error 26: Expected an expression, found ';'
badcode.c  19  Warning 522: Expected void type, assignment, increment or decrement
badcode.c  19  Error 10: Expecting ';'
badcode.c  19  Error 10: Expecting ';'
badcode.c  21  Info 718: printf undeclared, assumed to return int
badcode.c  21  Info 746: call to printf not made in the presence of a prototype
badcode.c  23  Info 737: Loss of sign in promotion from long to unsigned long
badcode.c  23  Info 713: Loss of precision (assignment) (unsigned long to long)
badcode.c  24  Error 2: Unclosed Quote
badcode.c  25  Error 10: Expecting ','
badcode.c  26  Error 10: Expecting ','
badcode.c  26  Error 26: Expected an expression, found '}'
badcode.c  26  Warning 559: Size of argument no. 2 inconsistent with format
badcode.c  26  Warning 516: printf has arg. type conflict (arg. no. 2 -- pointer vs. unsigned int)
with line 21
badcode.c  27  Warning 550: fellow (line 15) not accessed

--- Global Wrap-up
Warning 526: printf (line 21, file badcode.c) not defined
Warning 628: no argument information provided for function printf (line 21, file badcode.c)
Warning 526: fer (line 19, file badcode.c) not defined
Warning 628: no argument information provided for function fer (line 19, file badcode.c)
```

12. References

This is the full set of references used across the whole QuEST series. Not all the references are referred to in all of the QuEST papers. Most of these books have been reviewed by the ACCU. the reviews for these books and bout 3000 others are on http://www.accu.org/bookreviews/public/

Ball , Stuart. Debugging Embedded Microprocessor Systems, Newnes, 1998, ISBN 0-7506-9990-6

Baumgartner J Emulation Techniques, Hitex De internal paper, may 2001

Barr, Michael. Programming Embedded Systems in C and C++. O'Rilly, 1999, ISBN1-56592-354-5

Beach, M. Hitex *C51 Primer* 3rd Ed, Hitex UK, 1995, Beach, M. Hitex *C51 Primer* 3rd Ed, Hitex UK, 1995, http://www.hitex.co.uk (Draft 3.5 is on http://quest.phaedsys.org/)

Beach M, Embedding Software Quality Part 1, Hitex UK Available from www.Hitex.co.uk

Berger, Arnold. Embedded Systems Design: Anintroduction to Processes, Tools and Techniques. CMP Books, 2002, ISBN 1-57820-073-3

Black, Rex. Managing the Testing Process (2nd ed), Wiley, 2002, ISBN 0-471-22398-0

Brooks, Fred. The Mythical Man Month: Essays On Software Engineering, Anniversary Edition. Addison Wesley, 1995 ISBN 0-201-83595-9

Brown John, Embedded Systems Programming In C and Assembley, VNR, 1994, ISBN 0-442-01817-7

Buchner F Embedding Software Quality Part 1, Hitex DE Available from www.Hitex.co.uk

Buchner F The Classification Tree Method, Internal paper: Hitex DE, 2002

Buchner F The Tessy article for the ESC II Brochure Hitex DE, 2002

Burden, Paul. Perilous Promotions and Crazy Conversions in C, PR Ltd, MISRA-C Conference 2002. http://www.programmingreasearch.com/

Burns & Wellings Real-Time Systems and Their Programming Languages, Addison Wesley, 1989, ISBN 0-201-17529-0

Chen Poon & Tse, Classification-tree restructuring methodologies: a new perspective IEE Procedings Software, Vol 149 no 2 April 2002 pp 65-74

Clements Alan, 68000 Family Assembly Language Pub PWS 1994

Computer Weekly RAF JUSTICE :How the Royal Air Force blamed two dead pilots and covered up problems with the Chinook's computer system FADEC Computer Weekly 1997

Cooling J, Real-Time Software Systems ITC Press 1997 ISBN 1-85032-274-0

Cooling J. Software Design for Real time Systems ITC Press 1991 1-85032-279-1

COX B, Software ICs and Objective C, Interactive Programming Environments, McGraw Hill, 1984

Douglas BP Doing Hard Time, Developing Rea-Time Systems with UML, Addison Wesley, 1999, ISBN0-201-49837-5

Edwards, Keith. Real-Time Structured Methods: Systems Analysis, Wiley, 1993, ISBN 0-471-93415-1

Fertuck, L, Systems Analysis and Design with CASe tools Pub WCB 1992

Gerham, Moote & Cylaix, Real-Time Programming: A Guide to 32-bit Embedded Development, Addison Wesley, 1998, ISBN0-201-540-0

Hatton Les, *Safer C:Developing Software for High-integrituy and Safety Critical Systems*, Mcgraw-Hill(1994) ISBN 0-07-707640-0

Hills C A, Embedded C: Traps and Pitfalls Chris Hills, Phaedrus Systems, September 1999, quest.phaedsys.org/

Hills C A, *Embedded Debuggers* –Chris Hills & Mike Beach, Hitex (UK) Ltd. April 1999 http://www.hitex.co.uk & quest.phaedsys.org

Hills C A, Tile Hill Style Guide Chris Hills, Phaedrus Systems, 2001, quest.phaedsys.org/

Hills CA & Beach M, Hitex, **SCIL-Level** A paper project managers, team leaders and Engineers on the classification of embedded projects and tools. Useful for getting accountants to spend money Download from www.scil-level.org

HMHO Home Office Reforming the Law on Involuntary Manslaughter : The governments Proposals www.homeoffice.gov.uk/consult/lcbill.pdf

Johnson S. C. Johnson, *'Lint, a Program Checker,'* in *Unix Programmer's Manual*, Seventh Edition, Vol. 2B, M. D. McIlroy and B. W. Kernighan, eds. AT&T Bell Laboratories: Murray Hill, NJ, 1979.

Jones A History of punched cards. Douglas W. Jones Associate Professor of Computer Science at the University of Iowa.
http://www.cs.uiowa.edu/~jones/cards/index.html
see also http://www.cwi.nl/~dik/english/codes/punched.html

Jones, Derek. The 7+/- 2 Urban Legend. MISRA-C Conference 2002.
http://www.knosof.co.uk/

Kaner, Bach & Pettichord, Lessons Learned in Software Testing, A Context Driven Approach. , Wiley, 2002 ISBN 0-471-08112-4

Kernighan Brian W & Pike , The Practice of Programming. Addison Wesley 1999 ISBN 0-201-61586-X

Kerzner, Harold. Project Management: A Systems Approach to Planning, Scheduling, and Controlling. (7th ed) Wiley, 2001. ISBN 0-471-39342-8

Koenig A *C Traps and Pitfalls*, Addison Wesley, 1989

K&R *The C programming Language* 2nd Ed., Prentice-Hall, 1988
Lyons. JL, Ariane 5: Flight 501 Failure. Report by the Enquiry Board , Ariane, 1996

Maric B, How to Misuse Code Coverage. Reliable Software Technologies, 1997. www.testing.com

Maguire, Steve. Writing Solid Code, Microsoft Press, 1993, ISBN1-55615-551-4

McConnell Steve, Code Complete, A handbook of Practical Software Construction. Microsoft Press, 1993, ISBN 1-55615-484-4

MISRA Guidelines For The Use of The C Language in Vehicle Based Software. 1998 From http://www.misra.org.uk/ and http://www.hitex.co.uk/

Morton, Stephen. Defining a "Safe Code" Development Process, Applied Dynamics International, 2001

Murphy, Nial. Front Panel: Designing Software for Embedded User Interfaces, R&D Books 1998 ISBN 0-87930-528-2

Pressman Software Engineering A Practitioners Approach. 3rd Ed McGrawHill 1992 ISBN 0-07-050814-3

PRQA Programming Research QA-C static analysis tool.
www.programmingresearch.com

Randel, Brian. The Origins of Digital Computers, Springer Verlag 1973

Ritchie D. M. *The Development of the C Language* Bell Labs/Lucent Technologies Murray Hill, NJ 07974 USA 1993 available from his web site http://cm.bell-labs.com/cm/cs/who/dmr/index.html This is well worth reading.

Simon, David, An Embedded Software Primer, Addison Wesley,1999, ISBN 0-201-61569

Selis, Gullekson & Ward. Real-Time Object-Orientated Modeling, Wiley, 1994, ISBN 0-417-59917-4

Sutter Ed. Embedded Systems: Firmware Demystified, CMP Books, 2002 ISBN 1-57820-09907

Vahid & Givargis Embedded System Design: A Unified Hardware/Software Introduction, Wiley, 2002, ISBN 0-471-38678-2

Van Vilet Software Engineering Principals and Practice Pub Wiley 1993 ISBN 0-471-93611-1BN 0-471-93611-1

Watkins, John. A Guide To Evaluating Software Testing Tools (V3) Rational Ltd 2001

Watson & McCabe, Structured Testing: A testing Methodology Using the Cyclomatic Complexity Model

Webster, Bruce. The Art of Ware, Sun Tzu's Classic Work Reinterpreted, M&T Books, 1995 ISBN 1-55851-396-5

Whitehead, Richard. Leading A Software Development Team: A Developers Guide to Successfully Leading People and Projects, Addison Wesley, 2001 ISBN 0-201-67526-9

Wilson, Graham.. Embedded Systems & Computer Architecture, Newnes, 2002, ISBN 0-7506-5064-8

13. Standards

This is the full set of standards used across the whole QuEST series. These are Standards as issued by recognised national or international Standards bodies. Note due to the authors position in the Standards Process some of the documents referred to are Committee Drafts or documents that are amendments to standards that may not have been published by the time this is read.

ISO

9899:1990 Programming Languages - C

9899:1999 Programming Languages - C

9899:-1999 TC1 Programming Languages-C Technical Corrigendum 1

9945 Portable Operating System Interface (POSIX)
9945-1 Base Definitions
9945-2 System Interfaces
9945-3 Shell and Utilities
9945-4 Rational

12207:1995 Information Technology- Software Life Cycle Processes

14764:1999 Information Technology - Software Maintenance

14882:1989 Programming Languages - C++

15288:2002 Systems Engineering - System Lifecycle Processes

JTC1/SC7 N2683 Systems Engineering Guide for ISO/IEC 15288

WDTR 18037.1 Programming languages, their environments and system software
interfaces —Extensions for the programming language C to support embedded
processors

IEC

61508 :FCD Functional Safety or Electrical/Electronic/Programmable Electronic Safety -Relegated Systems

Part 1 General Requirements
Part 2 Requirements for Electrical/Electronic/Programmable
Electronic Safety -Relegated Systems

Part 3 Software Requirements
Part 4 Definitions and Abbreviations
Part 5 Examples of methods for the determination of SIL
Part 6 Guidelines for the application of parts 2 and 3
Part 7 Over View of Technical Measures

ISO/IEC JTC 1 N6981 Functional Safety and IEC61508: A basic Guide.

<u>IEEE</u>

You may be wondering where ANSI C is… ANSI C became ISO C 9899:1990 and ISO 9899 has been the International standard ever since. See "A Standard History of C" in Embedded C Traps and Pitfalls

1016-1998 Recommended Practice for Software Design Descriptions

5001:1999 The Nexus 5001 Forum™ Standard for a Global Embedded Processor Debug Interface

<u>NASA</u>

SEL-94-003 C Style Guide August 1994 Goddard Space Flight Centre

<u>OSEK</u>

Network Management Concept and Application Programming Interface
Version 2.50 31st of May 1998

Operating System Version 2.1 revision 1 13. November 2000

OIL: OSEK Implementation Language Version 2.2 July 27th, 2000

Communication Version 2.2.2 18th December 2000

<u>BCS</u>

Standard For Software Component Testing Draft 3.3 1997

PhaedruS
SystemS

Chris@phaedsys.org
http://www.phaedsys.org/